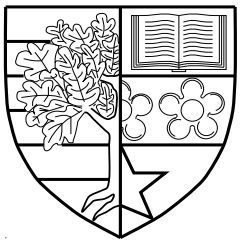
An Integrated Approach to High Integrity Software Verification

by

William James Ellis



Submitted for the Degree of
Doctor of Philosophy
at Heriot-Watt University
on Completion of Research in the
School of Mathematical and Computer Sciences
June 2010

Abstract

Computer software is developed through *software engineering*. At its most precise, software engineering involves mathematical rigour as *formal methods*. High integrity software is associated with safety critical and security critical applications, where failure would bring significant costs. The development of high integrity software is subject to stringent standards, prescribing best practises to increase quality. Typically, these standards will strongly encourage or enforce the application of formal methods.

The application of formal methods can entail a significant amount of mathematical reasoning. Thus, the development of automated techniques is an active area of research. The trend is to deliver increased automation through two complementary approaches. Firstly, *lightweight formal methods* are adopted, sacrificing expressive power, breadth of coverage, or both in favour of tractability. Secondly, *integrated* solutions are sought, exploiting the strengths of different technologies to increase automation.

The objective of this thesis is to support the production of high integrity software by automating an aspect of formal methods. To develop tractable techniques we focus on the niche activity of verifying *exception freedom*. To increase effectiveness, we integrate the complementary technologies of *proof planning* and *program analysis*. Our approach is investigated by enhancing the SPARK Approach, as developed by Altran Praxis Limited. Our approach is implemented and evaluated as the SPADEase system. The key contributions of the thesis are summarised below:

- **Configurable and Sound** - Present a configurable and justifiably sound approach to software verification.
- **Cooperative Integration** - Demonstrate that more targeted and effective automation can be achieved through the cooperative integration of distinct technologies.
- **Proof Discovery** - Present proof plans that support the verification of exception freedom.
- **Invariant Discovery** - Present invariant discovery heuristics that support the verification of exception freedom.
- **Implementation as SPADEase** - Implement our approach as SPADEase.
- **Industrial Evaluation** - Evaluate SPADEase against both textbook and industrial subprograms.

Acknowledgements

First and foremost, thanks to my supervisor, Andrew Ireland, who somehow managed to steer my erratic and plodding endeavours into something that warranted publication. Thanks are also due to my examiners, Greg Michaelson and Andrew McGettrick, for their balanced and constructive feedback.

Thanks go to Altran Praxis, in particular Peter Amey and Rod Chapman of SPARK Team, for their technical and moral support. Similarly, thanks are due to the Dependable Systems Group at Heriot-Watt University and the Mathematical Reasoning Group at Edinburgh University, for sharing their insights in an open and engaging manner.

Finally, gratitude is given to Christine Nichol and my parents. Through both infinite support and eternal nagging, they helped me get this finished.

This research was supported by EPSRC grants GR/R24081 and GR/T11289, and is a collaboration with Altran Praxis Limited.

ACADEMIC REGISTRY
Research Thesis Submission

Name*:*	William James Ellis		
School/PGI:	MACS		
Version: *(i.e. First, Resubmission, Final)*	Final	Degree Sought (Award **and** Subject area)	PhD, Computer Science

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1) the thesis embodies the results of my own work and has been composed by myself
2) where appropriate, I have made acknowledgement of the work of others and have made reference to work carried out in collaboration with other persons
3) the thesis is the correct version of the thesis for submission and is the same version as any electronic versions submitted*.
4) my thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librarian may require
5) I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.

* *Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is submitted.*

Signature of Candidate*:*		Date:	

Submission

Submitted By *(name in capitals):*	
Signature of Individual Submitting:	
Date Submitted:	

For Completion in Academic Registry

Received in the Academic Registry by *(name in capitals):*			
Method of Submission *(Handed in to Academic Registry; posted through internal/external mail):*			
E-thesis Submitted ***(mandatory for final theses from January 2009)***			
Signature:		Date:	

Please note this form should bound into the submitted thesis.

Updated February 2008, November 2008, February 2009

Contents

1 Introduction **1**

1.1 Motivation and Overview 1
1.2 Contributions 3
1.3 Publications 3
1.4 Thesis Organisation 4

2 Background **5**

2.1 Introduction 5
2.2 Program Verification 5
2.2.1 Axiomatic Assertional Reasoning 5
2.2.2 Abstract Interpertation 7
2.2.3 Program Refinement 7
2.2.4 Program Generation 8
2.3 Automated Deduction 8
2.3.1 Proof Assistants 8
2.3.2 Machine-Oriented Theorem Provers 9
2.3.3 Human-Oriented Theorem Provers 11
2.4 Program Analysis 11
2.4.1 Static Analysis 11
2.4.2 Invariant Discovery 12
2.5 Program Verification Systems 14
2.5.1 Batch Verification 14
2.5.2 Collaborative Verification 15
2.5.3 Lightweight Verification 16
2.6 Critical Analysis 19
2.6.1 Fundamental Positioning 19
2.6.2 Detailed Directions 20

3 Proof Planning **21**

3.1 Introduction 21
3.2 Proof Planning 21
3.2.1 Mathematical Reasoning 21

3.2.2 Proof Planning Architecture 22
3.3 Features of Proof Planning 24
3.3.1 Extensibility through Deep Understanding 24
3.3.2 Facilitates Sharing and Reuse 25
3.3.3 Constrained Search and Incompleteness 25
3.3.4 Flexibility through Separation of Concerns 25

4 The SPARK Approach 27
4.1 Introduction 27
4.2 SPARK Approach 27
4.2.1 Historical Perspective 27
4.2.2 Industrial Application 28
4.2.3 Overview 29
4.3 The SPARK Programming Language 30
4.3.1 Significant Language Features 30
4.3.2 Relationship to Ada 31
4.3.3 Example 32
4.4 The SPARK toolset 32
4.4.1 Conformance to SPARK 33
4.4.2 Data Flow Analysis 33
4.4.3 Information Flow Analysis 35
4.4.4 Program Verification 35
4.4.5 Partial Correctness 36
4.4.6 Exception Freedom 45
4.5 Configuring the SPARK toolset 49

5 Enhancing the SPARK Approach with SPADEase 50
5.1 Verifying Exception Freedom 50
5.2 Verification Challenges 51
5.3 Verifying Exception Freedom with SPADEase 52
5.3.1 Architecture of SPADEase 52
5.3.2 SPADEase Enhanced Verification Process 54
5.4 Addressing Verification Challenges 55

6 Proof Planner 57
6.1 Introduction 57
6.2 Proof Planner Architecture 57
6.3 Proof Planner Configuration 57
6.4 Illustrative Example 58
6.5 Goal 58
6.5.1 Declarations 60

6.5.2 Built-in Functions . . . 65
6.5.3 Goal Structure . . . 68
6.6 Theorems . . . 71
6.6.1 Retrieving Theorems . . . 71
6.6.2 Converting Theorems to Rewrite Rules . . . 72
6.7 Proof Plans . . . 75
6.7.1 Methods and Critics . . . 75
6.7.2 Strategies . . . 76
6.8 Proof Planner . . . 77
6.8.1 Plans . . . 77
6.8.2 Planner Algorithm . . . 79
6.9 Plan Result . . . 80
6.9.1 Result from Instantiated Proof Plan . . . 80
6.9.2 Result from Failure Critique . . . 81
6.10 Method-Language Overview . . . 83
6.11 Proof Plans Overview . . . 84
6.11.1 Proof Plans for Exception Freedom Goals . . . 84
6.11.2 Proof Plans for Program Analysis Queries . . . 86

7 Program Analyser **88**
7.1 Introduction . . . 88
7.2 Program Analyser Architecture . . . 88
7.3 Program Analyser Configuration . . . 90
7.4 MiniSPARK . . . 90
7.5 Parser . . . 90
7.5.1 Declarations . . . 90
7.5.2 Program . . . 91
7.6 Simplifications and Approximations . . . 96
7.6.1 Replace Named Scalar Constants With Their Values . . . 96
7.6.2 Eliminate Unneeded Casting . . . 96
7.6.3 Transform Return to Assignment . . . 96
7.6.4 Subprogram Call Abstractions . . . 96
7.7 Control Flowgraph . . . 97
7.7.1 Control Flowgraph Structure . . . 98
7.7.2 Subprogram Code as Control Flowgraph . . . 99
7.8 Structured Blocks . . . 101
7.9 Program Analyser Algorithm . . . 102
7.9.1 Program Analysis Methods . . . 102
7.9.2 Abstract Predicate Satisfiers . . . 103
7.10 Program Analysis Heuristics Overview . . . 103
7.10.1 Program Analysis Methods . . . 104

7.10.2 Abstract Predicate Satisfiers . . . 104

8 Evaluation 106

8.1 Introduction . . . 106
8.2 Implementation of SPADEase . . . 106
8.2.1 Implementing the Proof Planner . . . 106
8.2.2 Implementing the Program Analyser . . . 107
8.3 Evaluation of SPADEase . . . 107
8.3.1 Result Format . . . 107
8.4 Textbook Subprograms . . . 109
8.4.1 Subprogram Average . . . 109
8.4.2 Subprogram BubbleSort . . . 110
8.4.3 Subprogram DualFilter . . . 111
8.4.4 Subprogram MatrixFilter . . . 112
8.4.5 Subprogram MatrixMult . . . 113
8.4.6 Subprogram OpenPortScan . . . 116
8.4.7 Subprogram ResetArray . . . 118
8.5 Industrial Subprograms . . . 119
8.5.1 Subprogram 1 . . . 119
8.5.2 Subprogram 2 . . . 120
8.5.3 Subprogram 3 . . . 120
8.5.4 Subprograms 4 and 5 . . . 120
8.5.5 Subprogram 6 . . . 121
8.5.6 Subprogram 7 . . . 121
8.5.7 Subprogram 8 . . . 122
8.5.8 Subprogram 9 . . . 123
8.5.9 Subprogram 10 . . . 123
8.5.10 Subprogram 11 . . . 123
8.5.11 Subprogram 12 . . . 124
8.5.12 Subprogram 13 . . . 124
8.5.13 Subprogram 14 . . . 124
8.5.14 Subprogram 15 . . . 125
8.5.15 Subprogram 16 . . . 125
8.5.16 Subprogram 17 . . . 125
8.5.17 Subprogram 18 . . . 125
8.5.18 Subprogram 19 . . . 126
8.6 Overall Analysis . . . 126
8.6.1 Comparing Complexity and Proof Effort . . . 126
8.6.2 Comparing Complexity and Iterations . . . 129

9 Conclusions **130**
9.1 Contributions . . . 130
9.2 Limitations and Future Work . . . 132
9.2.1 Support Preconditions and Postconditions . . . 132
9.2.2 Adopt Tactic Based Theorem Prover . . . 133
9.2.3 Automated Lemma Discovery . . . 133
9.3 Summary . . . 133

A PolishFlag Interactive Proof **134**
A.1 Introduction . . . 134

B Modifying the SPARK Toolset **136**
B.1 Introduction . . . 136
B.2 Modifications to the Examiner . . . 136
B.3 Modifications to the Simplifier . . . 136
B.4 Modifications to the Checker . . . 136
B.4.1 Principled Proof Checking Interface . . . 137
B.4.2 Improve Predictability . . . 137
B.4.3 Richer Proof Commands . . . 137
B.4.4 Adding Theorems Through User Rules . . . 139

C Method-Language **141**
C.1 Introduction . . . 141
C.2 Method-Language Predicates . . . 141
C.3 Composition . . . 142
C.3.1 cut . . . 142
C.3.2 not . . . 142
C.4 Proof Planning . . . 142
C.4.1 abort_plan . . . 142
C.4.2 plan_lemmas . . . 142
C.4.3 write_line . . . 142
C.5 List Processing . . . 143
C.5.1 append . . . 143
C.5.2 filter_duplicates . . . 143
C.5.3 select . . . 143
C.6 Plan Features . . . 143
C.6.1 add_under_constrained_vars . . . 143
C.6.2 get_goal_category . . . 143
C.6.3 match_global_context . . . 144
C.7 Goal Features . . . 144
C.7.1 add_constraining_vars . . . 144

C.7.2 get_constraining_vars . . . 144
C.8 Goal Patterns . . . 144
C.8.1 aux_vars . . . 144
C.8.2 unconstrained_consts_vars . . . 145
C.8.3 uncoupled_entry_vars . . . 145
C.8.4 under_constrained_vars . . . 145
C.9 Analyse Expressions . . . 145
C.9.1 binary_explode . . . 145
C.9.2 conjunct_at . . . 145
C.9.3 elim_bounded_var . . . 146
C.9.4 eval_exp . . . 146
C.9.5 exp_at . . . 146
C.9.6 exp_explode . . . 146
C.9.7 find_replace . . . 147
C.9.8 ground . . . 147
C.9.9 int_bound_var . . . 147
C.9.10 is_inequality_op . . . 148
C.9.11 is_int . . . 148
C.9.12 prog_var_exps . . . 148
C.9.13 remove_real_exps . . . 148
C.9.14 replace_at . . . 148
C.9.15 simple_linear_exp_var . . . 148
C.9.16 solve_for_var . . . 149
C.9.17 sub_exp_polarity . . . 149
C.9.18 total_functions . . . 150
C.9.19 unconstrained_var . . . 150
C.10 Rewriting . . . 150
C.10.1 constants_to_value . . . 150
C.10.2 constrain_const_arrays . . . 150
C.10.3 constrain_exps . . . 150
C.10.4 eliminate_duplicate_vars . . . 151
C.10.5 select_alt_view_rule . . . 151
C.10.6 select_rewrite_rule . . . 152
C.10.7 select_transitivity_rule . . . 152
C.11 Rippling . . . 152
C.11.1 ripple_annotate . . . 152
C.11.2 ripple_complete . . . 152
C.11.3 ripple_erasure . . . 153
C.11.4 ripple_exp_at . . . 153
C.11.5 ripple_unblock_strategies . . . 153

C.11.6 select_wave_rule . . . 153

D Tacticals and Tactics **154**
D.1 Introduction . . . 154
D.2 Tactics . . . 154
D.2.1 null_tactic . . . 154
D.2.2 trivial_tactic . . . 154
D.2.3 trivially_true_conc_tactic . . . 155
D.2.4 rewrite_tactic . . . 155
D.2.5 split_conc_conj_tactic . . . 156
D.2.6 case_split_tactic . . . 157
D.2.7 sequence_tactic . . . 157
D.3 Tacticals . . . 157
D.3.1 then_tactical . . . 157
D.3.2 final_tactical . . . 157

E Proof Plans **158**
E.1 Introduction . . . 158
E.2 Proof Plans for Exception Freedom Goals . . . 158
E.3 Strategy: exception_freedom . . . 159
E.3.1 Behaviour . . . 159
E.4 Strategy: run_time_check . . . 160
E.4.1 Behaviour . . . 160
E.5 Strategy: invariant . . . 161
E.5.1 Behaviour . . . 161
E.6 Method: targeted_goal . . . 162
E.6.1 Behaviour . . . 162
E.7 Critic: proved_at_simplifier . . . 163
E.7.1 Behaviour . . . 163
E.8 Critic: simplified_goal . . . 164
E.8.1 Behaviour . . . 164
E.9 Critic: other_goal . . . 165
E.9.1 Behaviour . . . 165
E.10 Critic: in_real_domain . . . 166
E.10.1 Behaviour . . . 166
E.11 Method: initialisation . . . 167
E.11.1 Behaviour . . . 167
E.11.2 Introduce External Constraints . . . 167
E.11.3 Remove Duplicate Hypotheses . . . 168
E.11.4 Remove Real Hypotheses . . . 168
E.11.5 Replace Named Scalar Constants With Their Values . . . 168

E.12 Method: specialise_hyps . . . 169
E.12.1 Behaviour . . . 169
E.12.2 Plan Lemmas Separately . . . 170
E.13 Method: viable_goal . . . 171
E.13.1 Behaviour . . . 171
E.13.2 No Uncoupled Entry Variables . . . 171
E.13.3 No Unconstrained Constants or Variables . . . 173
E.13.4 No Under Constrained Constants or Variables . . . 173
E.14 Critic: couple_entry_vars . . . 178
E.14.1 Behaviour . . . 178
E.15 Critic: constrain_consts . . . 179
E.15.1 Behaviour . . . 179
E.16 Critic: constrain_vars . . . 180
E.16.1 Behaviour . . . 180
E.17 Critic: tightly_constrain_vars . . . 181
E.17.1 Behaviour . . . 181
E.18 Method: rtc_goal . . . 182
E.18.1 Behaviour . . . 182
E.19 Method: inv_goal . . . 182
E.19.1 Behaviour . . . 183
E.20 Method: true_conc . . . 183
E.20.1 Behaviour . . . 183
E.21 Method: false_conc . . . 184
E.21.1 Behaviour . . . 184
E.22 Method: linear_bounded_conc . . . 185
E.22.1 Behaviour . . . 185
E.23 Method: case_split . . . 187
E.23.1 Behaviour . . . 187
E.24 Method: mult_commute . . . 188
E.24.1 Behaviour . . . 188
E.25 Method: eval_conc . . . 189
E.25.1 Behaviour . . . 189
E.26 Method: split_conc_conj . . . 190
E.26.1 Behaviour . . . 190
E.27 Method: fertilize . . . 190
E.27.1 Behaviour . . . 191
E.28 Method: clear_conc_exp . . . 191
E.28.1 Behaviour . . . 191
E.29 Method: elim_var_conc . . . 192
E.29.1 Behaviour . . . 192

E.30 Method: transitivity_entry . . . 194
E.30.1 Behaviour . . . 194
E.31 Method: transitivity_decomp . . . 197
E.31.1 Behaviour . . . 197
E.32 Method: transitivity_fertilize . . . 199
E.32.1 Behaviour . . . 199
E.33 Method: transitivity_close . . . 201
E.33.1 Behaviour . . . 201
E.34 Method: transitivity_unblock . . . 202
E.34.1 Behaviour . . . 202
E.35 Method: ripple_entry . . . 203
E.35.1 Behaviour . . . 203
E.36 Method: ripple_wave . . . 206
E.36.1 Behaviour . . . 206
E.37 Method: ripple_fertilize . . . 208
E.37.1 Behaviour . . . 208
E.38 Method: ripple_unblock . . . 209
E.38.1 Behaviour . . . 209
E.39 Proof Plans for Program Analysis Queries . . . 210
E.40 Strategy: pa_exp_simplify . . . 211
E.40.1 Behaviour . . . 211
E.41 Strategy: pa_exp_constrain . . . 211
E.41.1 Behaviour . . . 211
E.42 Strategy: pa_spark_exp . . . 212
E.42.1 Behaviour . . . 212
E.43 Strategy: pa_disj_norm_form . . . 212
E.43.1 Behaviour . . . 212
E.44 Method: prune_conc_duplicate . . . 213
E.44.1 Behaviour . . . 213
E.45 Method: prune_conc_eq . . . 214
E.45.1 Behaviour . . . 214
E.46 Method: report_conc . . . 215
E.46.1 Behaviour . . . 215
E.47 Method: solve_eq_hyp_for_var . . . 216
E.47.1 Behaviour . . . 216
E.48 Method: constrain_conc_conj . . . 217
E.48.1 Behaviour . . . 217
E.49 Method: elim_aux_var_via_eq . . . 218
E.49.1 Behaviour . . . 218
E.50 Method: elim_prog_var_exp_via_eq . . . 219

E.50.1 Behaviour . . . 219
E.51 Method: elim_aux_var_via_int_arith . . . 220
E.51.1 Behaviour . . . 220
E.52 Method: is_spark_exp . . . 221
E.52.1 Behaviour . . . 221
E.53 Method: disj_norm_form . . . 222
E.53.1 Behaviour . . . 222

F MiniSPARK Grammar 223
F.1 Introduction . . . 223
F.2 Grammar . . . 223

G Program Analysis Methods 228
G.1 Introduction . . . 228
G.2 Method: scope . . . 228
G.2.1 Property Type . . . 228
G.2.2 Route . . . 228
G.2.3 Property Operations . . . 229
G.3 Method: update . . . 231
G.3.1 Property Type . . . 231
G.3.2 Route . . . 231
G.3.3 Property Operations . . . 231
G.3.4 Example . . . 234
G.4 Method: context . . . 239
G.4.1 Property Type . . . 239
G.4.2 Route . . . 239
G.4.3 Property Operations . . . 239
G.4.4 Example . . . 241
G.5 Method: type . . . 245
G.5.1 Property Type . . . 245
G.5.2 Route . . . 245
G.5.3 Property Operations . . . 245
G.6 Method: transient . . . 246
G.6.1 Property Type . . . 246
G.6.2 Route . . . 246
G.6.3 Property Operations . . . 246
G.6.4 Example . . . 248
G.7 Method: loop_range . . . 252
G.7.1 Property Type . . . 252
G.7.2 Route . . . 252
G.7.3 Property Operations . . . 252

G.8 Method: int_constraint 253
G.8.1 Property Type 253
G.8.2 Eliminate Expressions via Unconstrained Variables 254
G.8.3 Route 255
G.8.4 Property Operations 255
G.8.5 Example 261

Chapter 1

Introduction

1.1 Motivation and Overview

Computer software is increasingly prevalent in our modern society. This software can be roughly classified into *low integrity software*, where failure is irritating, and *high integrity software*, where failure brings significant costs. High integrity software may be found in safety-critical, security-critical and mission-critical contexts. Unfortunately, high integrity software failures do occur. Following seven billion dollars in development, a software error led to the destruction of Ariane 5 [Eur96]. A software error in the radiation therapy machine Therac-25 led to deaths from massive overdoses of radiation [LT93]. The Risks Digest [For] is updated frequently, describing recent vulnerabilities identified in computer systems and the risks they pose to the public.

Computer software is difficult to get right because it is inherently complex [Bro87, Ame01]. *Software engineering* [Som04] seeks to improve the quality of software by managing the processes under which it is developed. Software engineering may take many forms. At its most precise, software engineering is conducted with mathematical rigour as *formal methods* [CW96].

The advantage of formal methods is the additional leverage that mathematical rigour provides. Without any formality, software must be validated through *testing*. This involves checking that the software behaves correctly on a subset of the possible inputs. It is rarely practical to test all inputs, and thus testing can only offer a partial assurance that the software is correct [LS93]. With formality, software may be validated through *verification*. This involves formally verifying that the software meets its specification. Where applied in full, verification can give a complete assurance that the software is correct.

The development of high integrity software is subject to many standards [Int96, Min91, Rad93, Com98]. The standards aim to increase the quality of high integrity software by encouraging or enforcing best practises. In particular, for the most critical software, the Ministry of Defence Standard 00-55 [Min91] effectively mandates the use of formal methods [Tie92]. For these reasons, formal methods are commonly associated with the development of high integrity software. For example, in this context, there have

been several successful applications of formal methods in industry [BH97, CW96].

Despite the advantages offered by formal methods, their adoption remains marginal. Many of the criticisms directed at formal methods are based on flawed perceptions [Hal90, BH95a, BH95b, BH06]. Such misunderstandings may have arisen due to overly ambitious claims being made for formal methods [LG97]. Nevertheless, there are genuine obstacles in adopting formal methods. To facilitate migration to formal methods they should naturally extend existing software practises. However, advances in formal methods tend to occur as a new language or toolset with little emphasis on the engineering processes involved [FKV94]. Further, much of the tool support for formal methods has an academic background, and is not suited to industrial applications. Finally, The adoption of formal methods can require a significant learning process [WW93].

Recently, there has been interest in *lightweight formal methods* [JW96, AL98], accepting practical compromises to minimise the obstacles in adopting formal methods. The lightweight approach sacrifices expressive power, breadth of coverage, or both in favour of tractability. The strategy has been particularly successful by pursing *integrated* solutions, exploiting the strengths of various automated reasoning systems.

The objective of this thesis is to enhance the delivery of high integrity software by increasing automation in an area of formal methods. The aim is to develop tractable techniques by continuing the trend of lightweight formal methods. We investigate our approach within the context of the SPARK Approach [Bar03], as developed by Altran Praxis Limited. The SPARK Approach has been successfully applied in a wide range of high integrity software projects, including railway signalling, smartcard security and avionics systems [Cha00, BCJ+06]. We focus on the niche activity of verifying *exception freedom*. The SPARK Approach supports the verification of exception freedom [AC02] in the Floyd/Hoare assertional reasoning style [Flo67, Hoa69]. In this context, verifying exception freedom essentially involves verifying the absence of run-time errors [Ger78, GOC93]. Freedom from run-time errors is a key property desired of high integrity software. For example, a run-time error led to the loss of Ariane 5 [Eur96], and buffer overflows at run-time are the most common form of security vulnerability [CWP+00].

In verifying exception freedom there are two areas that may require manual interaction. Firstly, mathematical conjectures may need to be proved. Secondly, the specification of the program may need to be strengthened. Our approach aims to increase automation in both of these areas. The proof planning paradigm [Bun88] builds on mathematical intuitions to support automated deduction. It provides a flexible platform to develop proof automation strategies. Program analysis [NNH99] is a diverse field, enabling the automated extraction of information from programs. It provides a framework for automatically strengthening a program specification. We integrate proof planning and program analysis to create an automated program verification environment. In particular, we advocate a cooperative integration, with each component working together to more effectively deliver the required automation. Such an environment is tailored to offer increased

automation in verifying exception freedom in the SPARK Approach. Our approach is realised as the SPADEase system. This system is evaluated against industrial examples, with encouraging results.

1.2 Contributions

This thesis contains six main contributions in the field of automated program verification. The contributions are separated into three categories as listed below. The first category of contributions relate to the wider impact of our work:

- **Configurable and Sound** - Present a configurable and justifiably sound approach to software verification.
- **Cooperative Integration** - Demonstrate that more targeted and effective automation can be achieved through the cooperative integration of distinct technologies.

The second category of contributions relate to the specific processes developed in this work:

- **Proof Discovery** - Present proof plans that support the verification of exception freedom.
- **Invariant Discovery** - Present invariant discovery heuristics that support the verification of exception freedom.

Finally, the third category of contributions relate to the implementation of our work:

- **Implementation as SPADEase** - Implement our approach as SPADEase.
- **Industrial Evaluation** - Evaluate SPADEase against both textbook and industrial subprograms.

1.3 Publications

Aspects of this thesis have previously been presented in various different publications. For reference, each of these publications are listed below, highlighting their central contributions:

- **Workshop** ([IER02]) - We presented a one page position statement. We highlighted our intention to automate the verification of high integrity software by building upon the proof planning paradigm.
- **Conference** ([EI03]) - We presented a high level overview of our approach. At this stage, the essential ingredients of our approach were in place. We had focused on the niche activity of verifying exception freedom. Further, we tackled both proof

automation and specification strengthening through a proof planner and a program analyser respectively.

- **Conference** ([EI04]) - We presented key technical details of our approach. The form of our proof plans and program analysis heuristics are discussed. We observed the value of a collaborative integration in delivering automation. Further, we noted that our architecture enabled us to simultaneously accommodate both soundness and flexibility.

- **Conference** ([IEI04]) - We presented an offshoot from our primary work. The more general problem of verifying partial correctness is investigated. Similar to our exception freedom work, we address the challenge through a collaborative integration of proof planning and program analysis. Here, program analysis discovers contextual information as *invariant patterns* that are used to guide proof search. The ideas are illustrated through a worked example.

- **Journal** ([IEC+06]) - We presented substantial technical detail of our approach. Further, we described the favourable evaluation of our approach against a collection of industrial examples.

- **Workshop** ([JES07]) - We presented another offshoot from our primary work. We compare the capabilities of four automated reasoning tools in verifying exception freedom. The experimentation was supported through information and tools developed as part of this thesis.

1.4 Thesis Organisation

This introductory chapter provides a motivation for the research, highlighting its main contributions. Relevant background information is provided in Chapter 2. Greater detail on proof planning and the SPARK Approach is given in Chapter 3 and Chapter 4 respectively. A high level overview of SPADEase, as an enhancement of the SPARK Approach, is in Chapter 5. Details of the proof planner and its proof plans are presented in Chapter 6. Details of the program analyser and its program analysis heuristics are presented in Chapter 7. In Chapter 8 the evaluation of SPADEase on both industrial and textbook subprograms is reported. Finally, conclusions are made in Chapter 9.

Chapter 2

Background

2.1 Introduction

This chapter summarises the essential background for this thesis. Program verification is introduced in §2.2, focusing on automated approaches with immediate industrial applicability. Two fields closely associated with program verification are automated deduction, considered in §2.3, and program analysis, considered in §2.4. In §2.5, relevant program verification systems are described. Finally, a critical analysis of the background is made in §2.6, motivating the content of this thesis.

2.2 Program Verification

Program verification involves formally proving that a program conforms to its specification. There are several approaches that support program verification, including *axiomatic assertional reasoning* [Flo67, Hoa69], *operational assertional reasoning* [Moo06, MMRV06], *abstract interpretation* [CC77], *program refinement* [Bac78, Mor94], *program generation* [WH99a] and *program synthesis* [Bal85, Bie85, Gol86]. Here, we focus on approaches that readily admit automation and have already had an industrial impact.

2.2.1 Axiomatic Assertional Reasoning

Assertional reasoning was investigated by the early pioneers of electronic computing. Goldstine and von Neumann [GvN47] employed ‘assertion boxes’ to reason about the correctness of a program. Turing employed assertions in checking the correctness of an algorithm [Tur49]. These studies were exceptional and did not stimulate further research. However, they show that the value of program verification was quickly identified and suggest that assertional reasoning is an intuitive way to approach the task.

In assertional reasoning the semantics of the programing language and a complementary assertion language are formally defined. The program is annotated with assertions, specifying required properties at specific program points. Then, building on the formal

definitions, a reasoning process is undertaken to prove that the assertions always hold.

The first significant contributions to assertional reasoning were made independently by Floyd [Flo67] and Naur [Nau66]. Floyd introduced an *inductive assertion method*, demonstrating the feasibility of reasoning about computer programs. The method was not intended for practical use, being both involved and lacking scalability. However, the accessibility of the method stimulated further research in the area. The seminal paper by Hoare [Hoa69] extended the ideas of Floyd as an *axiomatic approach* to program verification.

The axiomatic approach introduced a simpler mechanism for reasoning about the behaviour of programs. Further, the approach is decompositional, being individually applied to each component of a program rather than the entire program. These features addressed scalability concerns, making the approach more tractable.

Hoare introduced what would become known as *Hoare-triples*, taking the form:

$$\{P\}\ S\ \{Q\}$$

Here, S is the statements of a component. P is a precondition and Q is a postcondition, specifying the required behaviour of the component. The triple is interpreted as a conjecture stating *partial correctness*. That is, if precondition P holds and the statements S terminate then the postcondition Q will hold.

Axioms support the decomposition of Hoare-triples, producing mathematical conjectures as *proof obligations* or *verification conditions* (VCs) in the process. The axiom for loops is particularly important and takes the form:

$$\frac{\vdash \{I \wedge G\}\ S\ \{I\}}{\vdash \{I\}\ \textbf{while}\ G\ \textbf{do}\ S\ \{I \wedge \neg G\}}$$

Here I is a loop invariant, a property that remains true for every iteration of the loop, and G is the loop guard. Significantly, a loop invariant must be provided for each loop to apply the axiom. Program analysis techniques may be able to automate the discovery of loop invariants. With loop invariants in place, the generation of VCs is entirely automatic. Proving the generated VCs proves the partial correctness of the program. Automated deduction techniques may be able to automatically discharge many of these VCs.

The axiomatic approach became the focus of a significant amount of further research [Apt81]. Various efforts were made to extend the axiomatic approach to accommodate additional programming language constructs. Further, the soundness and completeness of the approach was extensively investigated. In particular, Diskrta [Dij75] extended the approach through a *predicate transformer semantics*. This enabled proof of *total correctness*, showing that if the precondition holds then the program will terminate and the postcondition will hold.

2.2.2 Abstract Interpertation

Abstract interpretation [CC77] was introduced as a general framework for constructing program analysis systems. The program under analysis is *symbolically executed*, replacing concrete values with *abstract values* and concrete operations with *abstract operators*. The symbolic execution is iterative, strengthening and weakening the abstract values until a stable *fixed point* is reached. Termination is guaranteed, as the abstract values may always be approximated. The resulting *abstract model* may then be investigated accordingly.

The abstract values and abstract operators are customised to generate an abstract model of interest. The abstract model may be queried to report discovered properties [CH78]. Alternatively, the abstract model may be queried to highlight those properties that violate some given specification. Every genuine violation of the specification will be reported. However, as the technique is approximate, spurious violations may also be reported. Thus, each reported violation needs to be manually investigated to determine its validity. However, if all reported property violations can be dismissed as being spurious, then the analysis can be regarded as verifying that the specification holds.

The analysis is not decompositional, reporting on the characteristics of the entire system, rather than individual components. For example, a component may violate a specification when invoked with a certain combination of values. However, if the system never exercises this combination of values, then the component may still be reported as meeting the specification. Significantly, this lack of decomposition means that any change to the system invalidates all previous results. Further, as an analysis involves inspecting the entire system, efficiency is often a key issue.

Analysing a more complete system tends to increase the known constraints and improve the precision of the analysis. For this reason, abstract interpretation is typically applied retrospectively to completed systems. Further, to fully exploit system constraints, industrial grade applications of abstract interpretation are often specialised for the target system [SD07].

2.2.3 Program Refinement

Program refinement involves transforming a specification through a series of refinements until the specification becomes an executable program. By proving that each refinement is correct, the executable program must reflect the behaviour of the original specification. Program refinement is supported through a *refinement calculus* [Bac88, MV94, Mor94, Mor87]. The refinement calculus extends a programming language to include constructs that support the expression of specifications. A program refinement method will complement the refinement calculus with a supporting toolset. Tool support is required to ensure the correct application of refinement rules, maintain the details of each refinement step and discharge proof obligations. The potential for increased automation is recognised

[CR91, Nic93] and some progress has been made [Req08, BM99, BD96]. However, as the refinement process is inherently creative [KS04], a significant level of interaction is typically required.

2.2.4 Program Generation

Program generation involves automatically generating source code from a specification. Program generation supports two alternative approaches to program verification, as elaborated in [WH99a]. Firstly, program verification may be achieved by proving the correctness of the program generation system. For richer specifications, this can amount to a significant verification task. Consequently, this approach is typically associated with program generation systems that operate on low level specifications, such as compilers [Ste93]. Secondly, program verification may be achieved by conducting *translation validation*, proving that each generated program is correct [PSS98]. In general, it is much more tractable to verify the correctness of a generated program than the program generator itself. The richness of the specification language and the behaviour of the program generator may be configured to minimise the complexity of generated programs. Significantly, the source code produced by a generation system tends to exhibit a small collection of recurring patterns, making it particularly amenable to automated verification. Consequently, significant or completely automated translation validation is often feasible.

2.3 Automated Deduction

The automation of mathematical reasoning is known as *automated deduction*. Unsurprisingly, given the richness of mathematics and the wide range of potential applications, their are various different approaches to automated deduction. Following the classification of [Ker98], the approaches are considered according to their fundamental purpose in the sections below.

2.3.1 Proof Assistants

Proof assistants [Geu09] are geared toward assisting a mathematician in interactively completing a proof. The proof assistant behaves as a *proof checker*, verifying the correctness of each reasoning step, giving the mathematician confidence that their proof is correct. The proof assistant may also automatically discharge relatively trivial subgoals, allowing the mathematician to focus on the core proof problem.

The first proof assistant was Automath [NGdV94]. Automath employed an elegant *type theoretical* representation of mathematics, but acted as strict proof checker. Further type theoretical proof assistants were developed, such as Coq [Coq98] and LEGO [LP92], increasing the sophistication of the user interface and the level of proof automation. A key development was the introduction of *tactics* [GMW79]. A tactic is a subprogram that

manipulates the current goal. Various tactics may be introduced, providing the user with proof automation facilities. Several tactic based proof assistants have emerged, including Isabelle [Pau94], HOL [Gor88a], PVS [SORSC99] and Nuprl [C+86].

Despite some notable exceptions [Lov00], it is uncommon for mathematical proofs to be developed inside a proof assistant. One explanation for this is that there remains significant overheads in expression when comparing proof assistants with traditional mathematical texts.

2.3.2 Machine-Oriented Theorem Provers

Machine-oriented theorem provers adopt notations and algorithms that are particularly suited for mechanical processing. Typically, the internal behaviour of these provers are far removed from the mathematics they consider. Thus, the provers tend to be fully automatic, taking as input a conjecture and reporting a result as output.

There are two main challenges in developing machine-oriented theorem provers. Firstly, due to fundamental properties of computation [Koz97], it is not possible to construct automatic reasoning algorithms for all theories. Thus, a theory must be carefully selected which is expressive enough to be of practical use, yet simplistic enough to yield to automated analysis. Secondly, automatic reasoning algorithms are prone to suffer from a *combinatorial explosion* [Bun99], a rapid rise in computational overhead as conjecture complexity increases. Thus, extremely efficient data structures and algorithms are typically required.

One class of machine-oriented theorem provers are *decision procedures* [Koz97]. Decision procedures are restricted to expressively limited theories, equivalent to propositional logic. However, inside these theories, a decision procedure is able, in finite time, to determine the truth of a conjecture. There are many varieties of decision procedures, each targeting a carefully selected theory. For example, the Davis-Putnum procedure [DP60, DLL62] considers propositional calculus, Presburger arithmetic [Sta84] considers the natural numbers with addition and excluding multiplication, Ordered Binary Decision Diagrams (OBDDs) [Bry86] consider Boolean functions and finite-state automata have been used to consider the Weak Second-order Theory of one and two Successors (WS1S and WS2S) [Kla97]. Decision procedures have been extensively applied in hardware verification. However, due to the limited expressiveness of their theories, decision procedures have had a limited impact on program verification.

Another class of machine-oriented theorem provers are *semi-decision procedures*. Semi-decision procedures may operate in relatively expressive theories, equivalent to first order logic. However, inside these theories, a semi-decision procedure is only able to determine the truth of a conjecture in finite time if the conjecture is true. Initially, Herbrands theorem [Her30] demonstrated the feasibility of a semi-decision procedure for first order logic. Robinson made key efficiency savings to Herbrands theorem, as the resolution method [Rob65]. Several extremely efficient resolution based theorem provers have been

developed, such as Otter [McC94] and Vampire [RV02]. Famously, EQP proved a long standing mathematical conjecture that Robbins Algebras are Boolean algebras [McC97]. However, their often lengthy execution times, unfavourable semi-decidability, and wealth of configurable optimisations means they are less suited to the batch processing of numerous conjectures, as sought in program verification.

Some analysis tools may be regarded as specialised machine-oriented theorem provers. *Constraint solvers* receive as input a collection of constraints in a given theory, and seek to discover a satisfiable solution. For example, an integer constraint solver, when presented with:

$$(X > 1) \wedge (X < 5) \wedge (Y > 2) \wedge (Y < 10) \wedge (X + Y = Z)$$

might discover the satisfiable solution:

$$X = 2 \qquad Y = 3 \qquad Z = 5$$

The first constraint solvers emerged from *Constraint Logic Programming* (CLP) [JL87]. These have subsequently been generalised as *Constraint Programming* [Bar99], and employed in several niche applications [Wal96]. *Interval arithmetic* [Moo66, Hay03] is a mathematical technique that determines guaranteed bounds for a calculation. The technique may be used to reason about the precision of floating-point algorithms. *Model checkers* [CGP99] receive as input a model and properties that the model should meet. The model checker exhaustively explores the model state space, searching for a case where the properties do not hold. Model checkers are most commonly associated with hardware verification, as it has favourable state space characteristics. However, model checkers are increasingly used to complement software analyses. *Computer algebra systems* (CAS) support the automated manipulation of mathematical expressions. Theorem provers have been enhanced through an effective integration of such systems [KKS98].

A hybrid class of machine-oriented theorem provers are *SMT-solvers* (Satisfiability Modulo Theories) [BSST09, PBG05]. Such solvers exploit a combination of decision procedures alongside a collection of theories, such as arithmetic and arrays. By accepting richer theories, SMT-solvers are not decidable. However, in practice, the systems report conjectures as being provable, unprovable, or unknown in a timely fashion. Nelson and Oppen first introduced an architecture for combining decision procedures, as realised in their influential Simplify system [NO79]. Variations on this architecture have been investigated, increasing the sophistication of the integration between the decision procedures and the theories [FJOS03]. Richer theories, coupled with timely performance, means that SMT-solvers are particularly well suited to program verification. For this reason, SMT-solvers are an extremely active area of research. Notable systems include Yices [DdM06], CVC3 [BT07], and Z3 [dMB08].

2.3.3 Human-Oriented Theorem Provers

Human-oriented theorem provers adopt notations that are particularly suited for human understanding. The provers tend to be built from a large collection of mathematical heuristics, rather than a core reasoning algorithm. Significantly, where these heuristics fail, a mathematician should be able to comprehend the situation. They may be able to interactively complete the proof or even refine the heuristics such that a proof is found automatically.

An influential human-oriented theorem prover was Nqthm [BM88], also known as the Boyer-Moore theorem prover. The prover operated on a rich logic, syntactically expressed as Lisp [McC78]. Over decades, the prover was gradually enhanced with increasingly sophisticated heuristics. Many significant theorems have been successfully proved via Nqthm [BKM95]. However, to effectively use the prover, it is necessary to gain a strong understanding of its internal heuristics [BM90].

Bundy proposed an alternative paradigm for human-oriented theorem provers as *proof planning* [Bun88]. The paradigm makes a clear distinction between searching for a proof and checking the correctness of a proof. These two concerns are addressed in a consistent manner via *proof plans*, which are composed from *proof methods* and their supporting *proof critics*. Each proof method describes an intuitive reasoning step while its proof critics describe how to progress should this step fail. A key advantage of the paradigm is that proof plans expose the detail of controlling heuristics, supporting their scientific investigation [Bun91]. Several successful proof plans have emerged from *rationally reconstructing* the heuristics seen in the Nqthm prover. In particular, *rippling* [BBHI05] emerged from Nqthm heuristics for proof by induction. Three significant proof planning systems have been developed as the Clam provers [BvHHS90, RSG98], Omega [BCF+97] and IsaPlanner [DF03]. For further details on proof planning, see Chapter 3.

2.4 Program Analysis

The automated analysis of computer programs is known as *program analysis* [NNH99]. Two distinct classes of program analysis are considered in the sections below.

2.4.1 Static Analysis

Static analysers adopt a relatively closed architecture. The analysis is typically performed by a single, well-defined, algorithm. While the analysers always report results, the analysis undertaken may be relatively limited or approximative in general.

One of the first static analysers was Lint [Joh77], which sought to highlight potential ambiguities in C programs. Further *lint-like* static analysers were developed, each considering relatively low-level characteristics of the source code. Common analyses include *data use analysis*, *control flow analysis*, *interface analysis*, *information flow analysis* and

path analysis. The analyses tend to be computationally tractable, even on very large programs [ABC+07]. Further, the analysers often accept partial programs, as would be encountered during software development. The analysers operate by generating a report of their findings. The reports can be lengthy and may contain spurious errors.

Some static analysers operate by comparing their results against a specification of expected behaviour, and only report discrepancies. For example, Splint compares its analysis against provided assertions [LE01, EL02]. Further, the SPARK Approach compares calculated information flow against a provided assertion of expected information flow [Bar03, BC85]. While this style imposes an annotation burden, the analyser will only report genuine errors.

2.4.2 Invariant Discovery

A popular topic for program analysis research is developing approaches that offer effective invariant discovery. These approaches are classified into three strategies, as discussed in the sections below.

Guided by Additional Information

Invariant discovery may be assisted through additional information. Providing the additional information imposes a burden on the engineer. However, the burden may be significantly less than manually discovering invariants.

Dynamic analysers operate on the source code plus its associated test data. The source code is automatically instrumented to trace program values during execution. The program is subsequently executed on the test data, collecting the information trace. Through analysing the information trace, it is possible to discover program invariants. A key advantage of this approach is that many systems will already have a significant corpus of test data. However, the correctness of the invariants discovered are significantly dependent on the coverage offered by the test data. Further, as the analysis is one step removed from the code, those invariants discovered may not be directly relevant. An influential dynamic analyser was Daikon [EPG+07]. Daikon employs machine learning to discover probable program invariants. Daikon deduces abstract types present in the analysed program, supporting the filtering of less relevant properties [GPME06].

Predicate abstraction based program analysers operate on the source code of the program plus a collection of relevant program predicates. These predicates may be automatically calculated by other program analysis techniques or manually supplied by an engineer. Predicate abstraction [GS97] is a specialised form of abstract interpretation. Symbolic execution is replaced with the calculation of *strongest postconditions*. Consequently, the identification of a fixed point becomes the search for a loop invariant. In general, this may entail an infinite search, as loop invariants may be composed from an infinite number of potential predicates. Thus, to ensure termination, only the finite set

of provided predicates are explored. Significantly, it is thought to be easier to discover relevant predicates than to discover the loop invariants which are constructed from them. The technique has been successfully applied to reduce the invariant annotation burden in preforming program analysis [FQ02].

Heuristically Target Common Structures

Certain loop patterns occur more frequently in practice. As these patterns are identified, corresponding invariant discovery heuristics can be developed. On this basis, an extensive collection of invariant discovery heuristics have been proposed [EGLW72, KM73, Weg73, Weg74, GW75, Cap75, DM78, BBM97, Kov08]. From these studies, two general approaches have been identified. *Top-down* approaches begin by analysing the loop context, then working downwards toward the loop itself. *Bottom-up* approaches begin by analysing the loop itself, then working upwards towards the loop context. Typically, especially for rich invariant discovery, a combination of these approaches is necessary. While there is significant diversity in the invariant discovery heuristics proposed, a few broad techniques have emerged, as highlighted below.

A bottom-up heuristic, introduced in [EGLW72], involves solving *difference equations* or, more generally, *recursion relations*. A recurrence relation defines the n^{th} value of a sequence in terms of earlier values in the sequence. For example, consider the recurrence relation:

$$a_{(n)} = 2 * a_{(n-1)} + 1 \tag{2.1}$$

This describes that the n^{th} value of a is equal to twice the $(n-1)^{th}$ value of a plus one. A *solved recurrence relation* defines the n^{th} value of a sequence strictly in terms of n. For example, the recurrence relation above may be solved as:

$$a_{(n)} = 2^n * a_{(0)} + 2^n - 1 \tag{2.2}$$

Where an initial value is known, the general solution may be specialised. For example, if $a_{(0)} = 0$, the solution above may be specialised as:

$$a_{(n)} = 2^n - 1 \tag{2.3}$$

Significantly, recurrence relations may be used to express the value that a program variable will take on the n^{th} iteration of a loop. The solutions to such recurrence relations can be readily transformed into loop invariants. In general, the technique is relatively limited [Cap75]. However, for specific cases, in collaboration with further heuristics, useful invariants may be discovered [GW75, KM76, Kov08].

A top-down heuristic, introduced by Suzuki and Ishihata [SI77], is the *induction-iteration method*. Starting with a postcondition, the method works backwards through the

source code, seeking to calculate *weakest liberal preconditions*, the minimum constraints required to demonstrate partial correctness. Where encountering loops, an initial invariant is generated by calculating the weakest liberal precondition from the loop exit to the invariant cut-point. Typically, the initial invariant is flawed, not simultaneously supporting the verification of entering, iterating around and exiting the loop. A candidate refinement of the initial invariant is determined, by calculating the weakest liberal precondition from the initial invariant back to itself. The process may be repeated until a suitable invariant is generated. The technique may not terminate and may produce unnaturally verbose invariants. In [BLS96] the number of refinements is reduced through *refined strengthening*, minimising the verbosity of discovery invariants.

Proof Failure Analysis

Invariant discovery is typically undertaken to support program verification. Thus, the suitability of discovered invariants may be evaluated by their ability to support automated program verification. Where the verification effort fails, it may be subjected to *proof-failure analysis* to determine improvements to the discovered invariants. In [Ger78] invariant discovery is required in verifying the absence of run-time errors. Heuristics are employed to discover a candidate invariant, and the verification is attempted via a machine-oriented theorem prover. Where the verification fails, the invariant is strengthened to include those conclusions that could not be proved. Similarly, in [SI98], invariant discovery is required in verifying partial correctness. The postcondition is taken as an initial approximation for the invariant, and verification is attempted via a proof planner. The planner exploits the critics mechanism to introspect on any proof failures, suggesting invariant refinements. Through multiple iterations, the invariant may be refined to a stage such that the planner successfully completes the proof. Proof failure analysis tends to be involved, as it requires an effective integration of both invariant discovery heuristics and automated theorem proving. However, by considering the overall objective of program verification, the approach tends to produce relevant invariants.

2.5 Program Verification Systems

Over decades, several program verification systems have been developed. Three generations of systems are identified, and discussed in the sections below.

2.5.1 Batch Verification

The first generation of program verification systems sought to achieve an ambitious *batch verification*. In this style, once a program has been written, its verification is performed as a final step. These early program verification systems adopted the intuitive assertional reasoning approach.

The first system to demonstrate the feasibility of program verification was developed by King [Kin69]. The system targeted a simple programming language. Automated reasoning strategies were developed to assist in proving generated VCs. Further program verification systems were developed, targeting richer programming languages [Deu73, GLB75]. In these cases, VCs were manually proved inside a proof assistant. The Stanford Pascal Verifier [LGvH+79] was the first program verification system for a mainstream programming language. The SMT-solver Simplify [NO79] was employed to discharge VCs. Where Simplify was unsuccessful, a manual proof could be conducted inside a proof assistant.

2.5.2 Collaborative Verification

The batch verification systems demonstrated the feasibility of the assertional reasoning approach to program verification. However, these systems did not scale up to larger programs. Many argued that scalability could be achieved by performing a *collaborative verification* [Bac86, Dij76, Gor88b, Gri81, Kal90]. In this style, programs and their verification are developed simultaneously. By considering verification concerns during development, programs are more readily verifiable.

Recognising the need for a collaborative verification, further program verification systems were developed. Significant systems to emerge included the Gypsy Verification Environment (GVE) [AGB+77], AFFIRM [GMT+80] and the Hierarchical Development Methodology (HDM) [LNR80]. These systems took a broader view of program verification, enabling verification concerns to be developed during software development. The approach improved scalability, supporting the completion of a few significant verification efforts, primarily in the area of security related systems [SSDG81, Dev81, GSS82, KWAHT82, BKYH85, WLG+78, For80].

The Ada programming language [Ame83, Int95, Int07] was designed to replace the numerous programming languages being used at the United States Department of Defence. Ada has well defined semantics, making it particularly suited to program verification. Consequently, Ada based verification systems emerged, including Penelope [GMP90] and ANNA [LvHKBO87]. While these systems enjoyed academic success, they had little industrial impact. The SPARK Approach [Bar03] operates on a selected subset of Ada. SPARK has been successfully applied in niche areas of critical software development [Cha00]. For further details on the SPARK Approach, see Chapter 4.

Following advances in the assertional reasoning based approach to program verification, program refinement systems were investigated. Program refinement offers a naturally collaborative approach to verification, as the program literally emerges from the verification effort. Significant contributions include CIP [BBB+85, BEH+87], PROSPECTRA [HKB93] and B [Abr96]. The approach has had industrial successes employing B in the development of critical transport systems [GH90, HG93, BBFM99, BA05].

2.5.3 Lightweight Verification

The collaborative verification approach demonstrated verification of realistic software systems. However, its effective application required significant training and extensive tool support. To address these concerns there is a trend toward *lightweight verification*. In this style, expressive power, breadth of coverage or both is sacrificed to increase tractability. Several verification systems have been developed in this style, targeting different classes of verification.

Target critical specification

For a given system, certain areas of its specification may be particularly critical to its safe or secure operation. By targeting these critical areas the value of the verification effort is maximised. The trend is toward seamlessly extending mainstream programming languages with support for targeted verification.

Verification systems have been developed for the Java programming language. The LOOP system [vdBJ01] supports the verification of sequential, or non-threaded, Java programs. LOOP generates VCs by importing the code and specification into either the PVS or Isabelle proof assistant. The KeY tool [ABB+05] supports verification throughout the entire lifecycle of software development. Design and specification takes place in UML while implementation is in JavaCard. Verification facilities are integrated into a UML based computer aided software engineering tool (CASE). VCs are generated from the code and specification, and may be discharged automatically or interactively inside a theorem prover. These tools focus on providing an architecture that supports verification, allowing an engineer to concentrate on the genuine verification tasks of proof discovery and invariant discovery.

An obstacle to the adoption of program verification is that VCs are not intuitive to some software engineers. Thus, there is a trend toward isolating engineers from VCs, either through proof automation or alternative interfaces. Jive [MPH00] supports the verification of a subset of sequential Java. The program is reasoned about directly, through the interactive application of Hoare axioms, with any strictly logical conjectures being interactively discharged inside the PVS proof assistant. JACK [BRL03] supports the verification of Java applets. The code and specification are translated into the B system [Abr96], through the Atelier B [Cle] tool. Atelier generates VCs, and offers both automated and interactive proof. To ease interpretation of VCs, JACK automatically relates VCs to their corresponding code and specification. Krakatoa [MPMU04] supports the verification of sequential Java or JavaCard. The code and specification are exported into the Why verification tool [Fil03], supporting the generation of VCs. Similar to JACK, each VC is automatically related to the code and specification. Further, Why dispatches VCs to several automated theorem provers, automatically correlating their results. If desired, verification may be completed interactively.

The Microsoft .NET Framework supports the development of Microsoft Windows ap-

plications through various programming languages, including C#. To support program verification, the C# language was extended as Spec#, and the Spec# programming system [BLS05] was developed. Exploiting the technologies associated with the .NET Framework, the Boogie tool transforms Spec# code and specification into VCs. Abstract interpretation is employed to automatically discover program properties, including invariants. Further, VCs are dispatched to an automated theorem prover. The verification effort is presented strictly in terms of the source code and its specification [LMS05]. Thus, an engineer indirectly advances a proof by modifying the code or specification. An extension integrates Boogie with the HOL proof assistant [BLW08]. The extension enables VCs to be directly investigated and interactively proved.

Target critical behaviours

Classes of software systems may be particularly dependent on some critical behaviours. As these behaviours are generally applicable, the specification of their conformance may be automatically calculated. Further, by reasoning about specific behaviours, significant automation is often feasible. To maximise tractability, the trend is toward identifying behaviour violations, rather than proving their absence.

Model checking supports the identification of behaviour violations. Typically, behaviours of concurrent systems are investigated, such as deadlocks. The source code and the behaviours of interest are expressed as a model. Through model checking, behaviour violations are reported as counter-examples. The principle challenge is finding a compromise between accuracy and tractability. Java PathFinder [HP98] employs model checking to identify behaviour violations in Java bytecode. Scalability is tackled through heuristics that suggest effective model simplifications. The expectation is that the heuristics will be customised to suit a particular application or level of analysis. Bandera [HD01] applies model checking to identify behaviour violations in a large subset of Java. Bandera includes an integration of several program analysis components, which may be customised to effectively analyse a given system.

An alternative paradigm for targeted behaviour verification is *counter-example guided refinement* [CGJ+03]. The process begins by generating a sparse model of the program and its targeted behaviours, omitting details to increase tractability. The model is investigated via a model-checker to identify behaviour violations as counter-examples. The counter-examples are investigated, typically via a theorem-prover, to determine their validity. Where the counter-examples are valid, they are reported as behaviour violations. Otherwise, guided by the invalid counter-examples, the model is enriched to more accurately reflect the program behaviour, and the process is repeated. In principle, the analysis should terminate when no invalid counter-examples remain. In practice, due to tractability concessions and limitations of automated theorem proving, invalid counter-examples may remain. Systems based on this approach include SLAM [BBC+06] and BLAST [BHJM07]. These systems check that application program interfaces (APIs) are invoked

in a behavioural compliant manner.

Target absence of run-time errors

Run-time errors represent a common and critical defect in software. For a given language, the conditions under which run-time errors will occur are well defined. Thus, a specification for the absence of run-time errors can be automatically calculated. Further, given the relatively restricted nature of the problem, significant automation is often possible in verifying the absence of run-time errors.

Abstract interpretation systems can verify the absence of run-time errors. The Exception Analyser [WH99b] and PolySpace [Deu03] identify potential run-time errors in C, C++ or Ada code. The ASTRÉE Analyser aims to verify the absence of run-time errors of control software written in C that omit dynamic memory allocation and recursion. Most notably, ASTRÉE was specialised by its developers to verify that the primary flight control software of the Airbus A340 is free from run-time errors [BCC+03].

Assertional reasoning systems can verify the absence of run-time errors. The pioneering Runcheck system [Ger78] extended the Stanford Pascal Verifier to verify the absence of run-time errors in Pascal programs. Runcheck complemented the verification condition generator with a static analysis technique to reason about uninitialised variables [Ger81]. Further, Runcheck employed heuristics to discover invariants and discharged VCs with the SMT-solver, Simplify. The SPARK Approach [Bar03] supports the verification of *exception freedom* [AC02]. For the SPARK subset of Ada, this is essentially equivalent to verifying the absence of run-time errors. Similar to Runcheck, the verification condition generator performs static analysis to reason about uninitialised variables [BC85]. Limited type-based invariants are automatically inserted, and VCs are proved via the human-oriented theorem prover, the SPADE Simplifier. Caveat [ABC+94, RSB+99] was developed to verify the absence of run-time errors in control systems written in C that omit dynamic memory allocation. Caveat employs heuristics to discover invariants, and discharges VCs via a simplification tool. While all these systems offer automation, in practice, a realistic verification will likely require both manual proof discovery and invariant discovery.

The program generation system AUTOFILTER [WS04] transforms a high level description of a state estimation task into a C or C++ program. AUTOFILTER was customised to support translation validation, proving that the generated code met critical safety properties [DFS04]. In particular, these safety properties include proving that array-bound accesses will not lead to a run-time error. The generated program is submitted to an assertional reasoning system, generating VCs, which are proved in the theorem prover E-Setheo [MIL+97]. Constrained by code generation patterns and the safety properties of interest, invariant discovery is particularly tractable. Consequently, in practice, the verification effort is significantly automated.

Debugging tools have also been developed that highlight likely run-time errors. Sig-

nificantly, to increase both performance and automation, completeness and soundness are not strictly observed. Splint [LE01, EL02] identifies common programming errors in C programs. In particular, the approach identifies a common run-time error as buffer overflow vulnerabilities [Pet00]. Splint combines data flow analysis with a set constraint solver to perform its analysis. The Extended Static Checker for Java (ESC/Java) [FLL+02] identifies likely run-time errors in Java programs. For increased tractability, ESC/Java deliberately adopts unsound assertional reasoning, generating VCs and discharging these in an automated theorem prover. Interface changes have been proposed to ease interpretation of this imperfect analysis [KMD06]. Houdini supports an application of ESC/Java by automatically discovering numerous candidate invariants [FL01]. The candidate invariants are filtered by invoking ESC/Java and removing those that do not appear to be correct or relevant.

2.6 Critical Analysis

Here, the content of this thesis is motivated through a critical analysis of its related background. The fundamental positioning of the thesis is considered in §2.6.1 while its detailed directions are considered in §2.6.2.

2.6.1 Fundamental Positioning

Our motivation for this thesis is to enhance the development of high integrity software in industry. As discussed in §1.1, high integrity software development is subject to various standards. Many of these standards encourage the use of formal methods, including the application of program verification. We direct our attention at improving an aspect of program verification that is particularly relevant to high integrity software development.

As described in §2.5.3, there is a trend toward lightweight verification, placing greater emphasis on tractability than expressive power or breadth of coverage. A common feature of this approach is using a collaborative integration of existing technologies to deliver the required automation. Further, to support immediate applicability, the tendency is to enhance existing software processes. The trend has been successful, with systems such as ESC/Java and SLAM being routinely used in software development. Recognising its value to industry, we continue the trend of lightweight verification.

As explored in §2.2 and §2.5, several program verification approaches have been applied in industry. The axiomatic assertional reasoning approach is both intuitive and flexible. For these reasons, the approach is typically adopted where verification capabilities are retrospectively introduced into existing programming languages. The abstract interpretation approach is most effective when analysing completed systems. However, as highlighted in §2.5.2, verification is more readily achieved when considered throughout software development. Program refinement naturally supports the progression of verification during software development. However, the approach lacks flexibility, requiring the

adoption of a specific development process and supporting toolset. Program generation offers significant potential for automating both software development and its verification. However, the automation is achieved by focusing on a specific application domain. Seeking immediate industrial applicability, we favour a verification approach that complements existing software development processes. Further, to increase effectiveness, we favour an approach that allows verification concerns to be considered during development. For these reasons, we chose to investigate program verification based on axiomatic assertional reasoning.

As observed in §2.5.3, verification in the SPARK Approach is achieved through axiomatic assertional reasoning. The SPARK Approach has been successfully applied in developing high integrity software, providing a fitting framework for investigating our research. Further, the SPARK Approach supports the targeted verification of exception freedom. Such a constrained verification activity closely aligns with our objective of investigating lightweight verification.

2.6.2 Detailed Directions

In the SPARK Approach, verifying exception freedom is essentially equivalent to verifying the absence of run-time errors. As observed in §2.5.3, a number of systems support verifying the absence of run-time errors. The configuration offered by these systems is limited. As observed in §2.2.2, the success of abstract interpretation can depend on application specific specialisations. Making such specialisations requires considerable technical skill, customising the underlying abstract model and verifying its correctness. However, this demonstrates the potential value of configuration. On this basis, we pursue an architecture that offers tractable configuration while simultaneously preserving soundness.

Verifying the absence of run-time errors requires both proof discovery and invariant discovery. The Runcheck, Caveat and ESC/Java systems consider these related tasks as separate activities. As discussed in §2.4.2, proof failure analysis supports effective invariant discovery in verifying partial correctness. Thus, we investigate the use of proof failure analysis to guide invariant discovery for the constrained task of verifying exception freedom.

Although a number of systems verify the absence of run-time errors, few report results for high integrity software. By investigating our techniques within an existing high integrity software development process, there is the potential for industrial evaluation. Thus, we aim to evaluate our approach against both textbook and industrial subprograms.

Chapter 3

Proof Planning

3.1 Introduction

This chapter describes the proof planning paradigm. The motivations behind proof planning and its supporting architecture are presented in §3.2. Significant features of proof planning are discussed in §3.3.

3.2 Proof Planning

The aim of a *science of reasoning* [Bun91] is to understand and document the processes involved in reasoning. While a science of reasoning applies to all forms of reasoning, emphasis is typically placed on mathematical reasoning. Proof planning [Bun88] describes an architecture which supports the automated and scientific investigation of mathematical reasoning. Proof planning builds on fundamental observations about mathematical reasoning. Thus, it is beneficial to consider mathematical reasoning before describing the architecture of proof planning.

3.2.1 Mathematical Reasoning

The discovery of mathematical results is achieved through mathematical reasoning. Despite limited research in this area, a few broad observations about mathematical reasoning may be made.

Proof Discovery

According to Polya [Pol54], the process of mathematical reasoning is performed as two different tasks, as summarised below:

- **Plausible reasoning** - Mathematicians approach a new problem with plausible reasoning. Relying on their intuitions the mathematician sketches out a plausible proof. While the proof is plausible, it may be flawed in practice.

- **Demonstrative reasoning** - Guided by a plausible proof, mathematicians employ demonstrative reasoning. The plausible proof is rigorously investigated to generate a demonstrative proof of correctness.

Proof Families

Plausible reasoning is achieved through mathematical intuitions. Such intuitions arise due to *proof families*, similar problems that are susceptible to similar proofs. Many proof families have been documented in mathematics. Common patterns have been identified in limit theorems [BBH72], finding fixed-point combinators [WM88] and compass constructions [Pol65]. Rippling [BBHI05] exploits a structural pattern to guide a proof by induction. The general mechanism of rippling has also been applied to summing series [WNB92], conjectures with multiple induction hypotheses [YBG+94], logical frameworks [NY96] and general equational reasoning [Hut97].

Proof Languages

Polya [Pol65], and later Bundy [Bun91], observe that plausible reasoning and demonstrative reasoning are undertaken in two contrasting languages:

- **High level explanation language** - Plausible reasoning is described in an informal explanation language, suitable for describing the story of a proof.
- **Logical language** - Demonstrative reasoning is described in a formal logic, providing an unambiguous and exhaustive description of a proof.

Proof Understanding

Robinson [Rob97] argues that plausible reasoning provides an *explanation* while demonstrative reasoning provides a *guarantee*. Significantly, the two products are regarded as being distinct. It is possible to have correct intuitions about how a proof will proceed, without appreciating the step by step details of a logical proof. Conversely, it is possible to have a detailed logical proof without appreciating the fundamental principles being employed. Consequently, to fully understand a proof, both the explanation and guarantee are required:

$$Proof = Guarantee + Explanation$$

3.2.2 Proof Planning Architecture

Since its inception in [Bun88], various extensions and refinements of proof planning have been proposed. A consequence of these modifications is that there is not a uniform definition of proof planning [Den05]. In this thesis we focus on the original description of proof planning in [Bun88], extended to accommodate critics [Ire92], as implemented in

Clam [BvHHS90]. A detailed discussion of alternative perspectives on proof planning can be found in [DJP06].

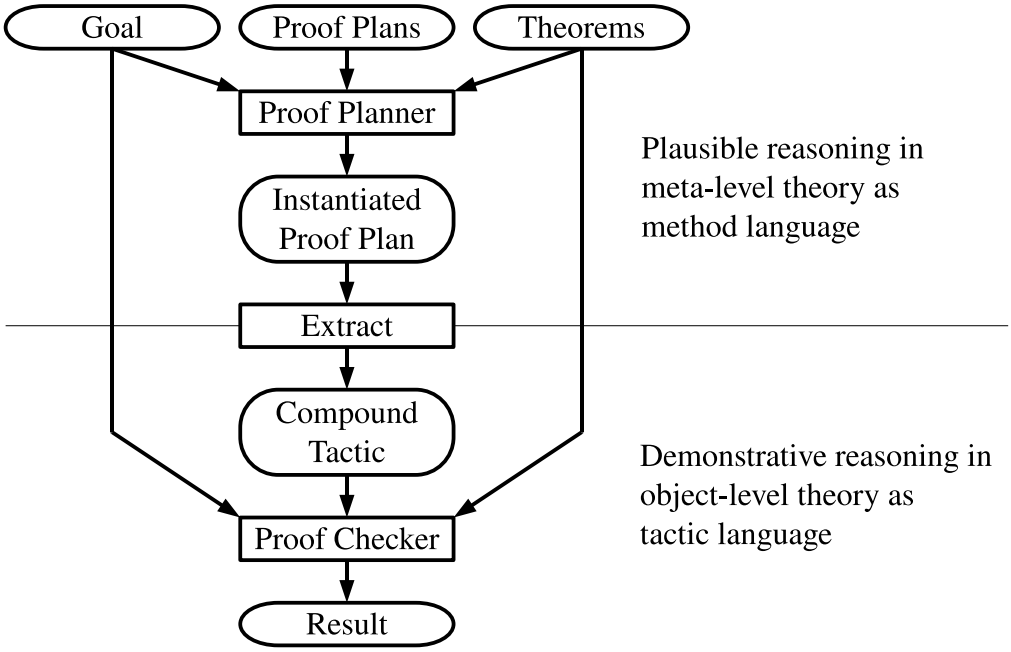


Figure 3.1: Proof planning architecture

The proof planning architecture is shown in Figure 3.1. Reflecting mathematical reasoning, proof discovery is separated into the two tasks of plausible reasoning and demonstrative reasoning. Appropriate proof languages are employed for these tasks as described below:

- **Meta-level theory** - The meta-level theory supports the expression of plausible reasoning. The language is captured as a flexible *method-language*. The method-language is heuristic by nature, being extended and refined as proof plans are developed.

- **Object-level theory** - The object-level theory supports the expression of demonstrative reasoning. The language is captured as *tactics* [GMW79]. The tactics are subprograms that perform logical transformations.

Plausible reasoning is achieved by a *proof planner*. The proof planner is provided with a *goal*, *theorems* and *proof plans*. The theorems describe properties and definitions that are valid for the goal. The proof plans capture mathematical intuitions behind a family of proofs, as described below. The proof planner searches for a proof of the goal in the meta-level theory, guided by the proof plans and appealing to theorems as necessary. The proof planner explores the search space via a proof tree. Common search strategies include depth-first, breath-first and iterative deepening. The proof plans may also influence the search strategy.

Demonstrative reasoning is achieved by a *proof checker*. Where successful, the proof planner will discover an *instantiated proof plan*. While proof plans describe a family of proofs, an instantiated proof plan describes the proof of a particular goal in a particular context. A *compound tactic* is extracted from the instantiated proof plan. The compound tactic is a sequence of tactic applications describing a logical proof of the goal. The compound tactic is submitted to a proof checker, alongside the original goal and theorems. The proof checker is a sound tactic based theorem prover. Driven by the compound tactic, the proof checker checks the validity of the discovered proof. The proof is valid if the proof checker discharges the goal.

Proof Plans

Proof plans are at the core of proof planning. A proof plan captures the mathematical intuitions behind a family of proofs. To encourage this ideal, favourable and measurable criteria of proof plans have been explicitly identified, as described in [Bun91]. Proof plans are expressed as *proof methods* and *proof critics*. Typically, a proof plan is composed from several methods and critics.

Proof methods aim to advance the proof of a goal. The applicability of a method is constrained through preconditions expressed in the method-language. The effect of the method is described in both the meta-level theory and the object-level theory through the method-language and tactics respectively. Two different types of methods may be identified. A non-terminating method transforms the goal into one or more subgoals. A terminating method eliminates a trivial goal.

Each proof critic is associated with a proof method. The critic seeks to recognise and patch common patterns of method failure. A critic is triggered when its corresponding method has a particular pattern of precondition failure. Like methods, the applicability of a critic is constrained through preconditions expressed in the method-language. The effect of the critic is only described in the meta-level theory. A critic may have any effect, ranging from local changes to a single goal through to global changes to the entire proof.

3.3 Features of Proof Planning

By mirroring mathematical reasoning, proof planning has many valuable features, as discussed in the sections below.

3.3.1 Extensibility through Deep Understanding

Unlike many other automated reasoning paradigms, proof planning places an emphasis on proof understanding over proof automation. This shift in emphasis affects the properties of the resulting automated reasoning system. Where focusing on proof automation, there is initially rapid progress. Heuristics are introduced in reaction to unproven conjectures.

As the number of heuristics increase, they will inevitability start to clash. Eventually these clashes hinder further development. Where focusing on proof understanding, there is initially slow progress. Heuristics are only introduced as genuine reasoning patterns are discovered. As each heuristic captures a coherent portion of reasoning, they are naturally cooperative. Thus heuristics may continue to be introduced without hindering further development. While this is a comparison of two extremes, the general trend is valid. Proof planning leads to a more extendable reasoning system through deeper proof understanding. The cost is that such a rigorous approach requires greater effort to develop.

3.3.2 Facilitates Sharing and Reuse

Proof plans are expressed through external components in a uniform style. Presenting proof plans in this manner means that they are readily accessible by the automated reasoning community. Further, proof plans make a clear distinction between proof search and proof checking. Thus, it is relatively straight forward to reuse the proof search portion of proof plans in different logical domains. For example, in [IS00], proof plans developed for mathematical induction are reused in automating the discovery of loop invariants.

3.3.3 Constrained Search and Incompleteness

A proof planner conducts its search in the meta-level theory. The meta-level theory offers a higher level of abstraction than the object-level theory. A proof step at the meta-level, for example *simplify*, might correspond to a number of proof steps at the object-level. Thus, by searching in the meta-level theory, a smaller search space is explored. A consequence of searching at the meta-level theory is losing completeness. The proof steps of the meta-level theory may omit valid proof steps in the object-level theory. This weakness can be minimised by developing principled proof plans. In this case, any loss of completeness corresponds to a missing proof plan.

3.3.4 Flexibility through Separation of Concerns

Proof planning makes a clear distinction between proof search and proof checking. Proof search is performed in a proof planner while proof checking is performed in a proof checker. Significantly, soundness depends solely on the proof checker. Any reasoning error introduced at proof search will be detected and rejected during proof checking. The architecture frees proof search from the burden of demonstrating soundness, supporting the flexible development of sophisticated heuristics. Two general techniques that exploit this flexibility are detailed below.

Contextual Information

The description of a goal may be supplemented through contextual information. The information is typically meta-logical, being relevant to the goal yet not directly expressible as part of the goal. Heuristics can exploit such information to offer a more targeted proof search [DJP06]. For example, program verification is considered in [IEI04]. VCs are supplemented with contextual information that reveals proprieties of the corresponding program. The domain knowledge is exploited to constrain proof search. Contextual information is often embedded into a goal through annotations. For example, rippling [BBHI05] is controlled through expression annotations, and a general formalism of annotations has been developed [HK97].

Middle-Out Reasoning

In general, the proof of a goal is advanced by applying a favourable transformation. In many instances the goal and its preceding context provides little guidance in selecting the next transformation. Resolving such blocking points require a creative *eureka* step. Typically, the merit of a eureka step only becomes apparent later in the proof.

A successful strategy for discovering eureka steps is *middle-out reasoning* [BSH90], which builds on ideas originally developed in GPS [EN69]. The strategy exploits the observation that the merit of a eureka steps often becomes apparent later in a proof. Thus, where a eureka step is required, its choice is delayed and the proof is continued. The intention is that the structure of the continued proof will introduce additional constraints, revealing the shape of the eureka step. Essentially, the strategy develops the middle of a proof to gain deeper insights into an earlier stage of the proof.

Middle-out reasoning requires a mechanism to delay the selection of a eureka step. This is achieved by replacing a eureka step with a meta-variable. Meta-variables range over all valid expressions, thus they simultaneously represent every possible transformation. With the meta-variable in place, the proof may continue. As the proof progresses, the meta-variable will be incrementally instantiated. Where the meta-variable becomes fully instantiated it will reveal the form of the eureka step.

Middle-out reasoning has significant implications for proof search. The search is radically reduced by simultaneously considering all possible transformations through the introduction of a meta-variable. However, the presence of a meta-variable will significantly increase the applicability of proof steps. For this reason, middle-out reasoning is typically only practical where strong expectations about the proof are known. These expectations can be exploited to constrain the search, selecting a few promising proof steps from the numerous applicable proof steps.

Chapter 4

The SPARK Approach

4.1 Introduction

This chapter describes the SPARK Approach. In §4.2 the features of the approach are summarised. The SPARK programming language and the SPARK toolset are described in §4.3 and §4.4 respectfully.

4.2 SPARK Approach

The SPARK Approach addresses the specific challenge of developing high integrity software. Here, the background of the approach is presented, summarising its history and significant industrial applications. Following this, an overview of the approach is given, describing its key attributes.

4.2.1 Historical Perspective

The origins of the SPARK Approach may be traced back over twenty five years, to program analysis research undertaken at Southampton University. Key products of this research included the mathematical foundations of a form of information and data flow analysis [BC85] and development of the SPADE (Southampton Program Analysis and Development Environment) toolset [O'N87, CCDO86]. The SPADE toolset supports the analysis of Assembly language programs for the 68020 and Z8002 processors, the 8096 Intel microcontroller and programs written in a subset of Pascal. Program Validation Limited (PVL) was established to support the commercialisation of this research. At PVL, the SPARK (SPADE Ada Kernel) programming language was defined as a formal subset of Ada [CG90, Mar94, O'N94]. Building upon the SPADE toolset, tools were developed to support the analysis of SPARK programs. Further, the possibility of proving that SPARK programs were free from run-time exceptions was investigated [GOC93]. Following these achievements, PVL was acquired by Praxis Critical Systems Limited (Praxis-CS). The larger infrastructure at Praxis-CS enabled the techniques developed at PVL to be applied

on large scale high integrity software projects. In tackling larger projects the techniques were refined and strengthened accordingly. The verification of exception freedom was transformed from a theoretical possibility to a practical reality [AC02]. Further, support for concurrent applications was introduced as RavenSPARK [AD03]. Learning through experience, these larger projects inspired guidelines for the application of SPARK. These guidelines, together with the SPARK language and supporting toolset, formed the basis of a complete approach for developing high integrity software. The maturity of the approach was signalled with the publication of the SPARK Approach book [Bar03]. Praxis-CS merged with High Integrity Systems Limited (HIS) to become Praxis High Integrity Systems Limited (Praxis-HIS). The profile of SPARK was increased through a technical and marketing partnership with AdaCore. In particular, SPARK is now licensed under the GNU General Public License (version 3) [GNU]. Following a merger with SC2, Praxis-HIS become Altran Praxis Limited.

4.2.2 Industrial Application

The SPARK Approach has been successfully applied in several high integrity software projects. To illustrate the industrial applicability of the approach, a collection of these projects are summarised below.

- **C130J MC** - The Mission Computer (MC) is a critical avionics system at the core of the Lockheed C130J, a military and commercial transport aircraft. The dual application of the aircraft means that the MC is subject to a number of standards, including DO178B [Rad93]. The MC had already been specified following the CoRE technique [FFK94]. Working from this specification, a standard compliant MC was successfully implemented following the SPARK Approach [CS95, Cha00].

- **MULTOS CA** - The Certification Authority (CA) is a security critical component of the Multi-Application Operating System (MULTOS) intended for use on smartcards. To attain the security confidence demanded, the CA component needed to meet the highest level (E6) of the Information Technology Security Evaluation Criteria (ITSEC) [Com98]. The SPARK Approach was successfully employed in designing and implementing the CA to the security level required [HC02, Cha00].

- **SHOLIS** - The Ship Helicopter Operational Limits Instrumentation System (SHOLIS) resides on a ship to provide information about the safe use of helicopters in various situations. Given the nature of the system, it was subject to Defence Standard 00-55 for safety critical software [Min91]. The SPARK Approach was successfully employed in specifying, designing and implementing SHOLIS [Cha00]. The resulting system was the first to meet every requirement in the stringent Defence Standard 00-55 for safety critical software.

- **Tokeneer** - The Tokeneer ID Station (TIS) [Tok] was developed by the National Security Agency (NSA) to investigate access control and biometrics. In this instance, the TIS served as a demonstrative system as part of a controlled experiment to evaluate the overall effectiveness of the SPARK Approach. The system had to meet Evaluation Assurance Level 5 (EAL5) of the Information Technology Security Evaluation Criteria [Com98]. The SPARK Approach was employed in specifying, designing and implementing the TIS system. The SPARK Approach successfully developed the system to the desired assurance level in a cost effective manner. In practice, due to the inherent rigour of the SPARK Approach, some assurances could be made beyond EAL5. Following the experiment, it was speculated that the SPARK Approach would also be able to effectively deliver at EAL7, the highest assurance available under the Common Criteria [BCJ+06].

More generally, the merits of the SPARK Approach have been recognised by independent organisations that are concerned with the development of high integrity software. Following the successful Tokeneer project, the US National Cyber Security Partnership highlighted the SPARK Approach as one of only three development processes able to deliver sufficient assurance for security critical systems [Nat04]. The US National Academies [JTM07] advocates the use of simple, well defined, and safe programming languages, especially where developing critical applications. In this context the SPARK Approach is referenced, highlighting its industrial successes. The US Defence Technical Information Centre [GWM+07] references the SPARK Approach, in the context of applying formal methods to develop secure software systems.

4.2.3 Overview

The SPARK Approach has matured into a complete discipline for developing high integrity software. The guiding philosophy of the approach is Correctness by Construction (CbyC) [Ame06, HC02, Ame01, Bar03]. Essentially, the central premise of CbyC is to build software right to begin with, rather than embark on a lengthy and costly process of identifying and eliminating errors. While it is accepted that defects will inevitably occur during development, significant progress can be made by *striving* for zero defects and selecting notations and tools accordingly.

To meet the objectives of CbyC it is essential that the philosophy is pursued throughout the entire lifecycle of software development. To this end, the SPARK Approach offers guidance at each key stage of the lifecycle as summarised below.

- **Requirements and Specification** - The requirements and specification is elicited through REVEAL [Pra01]. The REVEAL method provides guidance on effectively addressing the various concerns seen in requirements engineering. A key concern identified in REVEAL is the selection of notations to ensure unambiguous require-

ments. Where following the REVEAL method it is not uncommon for formal notations, such as Z [Spi92], to be adopted.

- **Design** - Design is guided through the Informed method (INformation Flow Oriented MEthod of Design) [Ame99, Bar03]. Informed reinforces the merits of *low coupling* and *strong cohesion*. In particular, it argues that these favourable properties can be attained by positioning state to minimise information flow.

- **Implementation** - Implementation is achieved through the SPARK programming language and the SPARK toolset. The SPARK programming language is a formal subset of Ada, as described in §4.3. The SPARK toolset supports various analyses of SPARK programs, as described in §4.4.

4.3 The SPARK Programming Language

The SPARK programming language is at the centre of the SPARK Approach. Significant features of the language are discussed below, including the relationship between SPARK and Ada. Finally, for illustration, a small example is presented.

4.3.1 Significant Language Features

At its inception, key requirements for the SPARK programming language were established. The design of the SPARK language continues to be guided by these requirements:

- **Logical soundness** - It must be possible to reason precisely about the semantics of the programming language. Thus the language must be logically sound.

- **Simplicity of language definition** - A simple programming language is easier to understand than a complex one. Improved understanding decreases the likelihood of errors being made. Thus, simplicity is sought in the language definition.

- **Expressive power** - A practical programming language must have sufficient expressive power to support the development of realistic applications. Thus, simplicity must always be balanced against expressiveness.

- **Security** - Language insecurity occurs when a program breaks the rules of the language at run-time. Thus, it must be possible to statically demonstrate that a program is secure.

- **Verifiability** - The programming language must be amenable to program verification.

- **Correspondence with Ada** - Costly compiler development can be avoided by expressing the language as a pure subset of Ada. Significantly, Ada is sufficiently

formal such that this compiler reuse does not overly compromise other language requirements.

- **Verifiability of compiled code** - It must be possible to demonstrate that a compiled program faithfully reflects the semantics of its source code. The language should favour constructs that generate more readily verifiable object code.
- **Bounded space and time** - It must be possible to show that a program operates within fixed space and time requirements. The language should exclude constructs that hinder calculating maximum memory usage and worst-case execution times.
- **Complexity of run-time system** - Run-time libraries may need to be subjected to the same level of certification as the application program. Thus, to ease certification, the language must be able to operate with little, or zero, run-time library support.

The SPARK language has emerged from a careful balancing of the above requirements. Significant features of the language are highlighted below:

- **Contracts** - The required behaviour of subprograms may be specified through explicit and verifiable contracts.
- **Structured control flow graph** - Various language restrictions are imposed to ensure that the program has a well-structured, and thus readily analysable, control flow graph.
- **No pointers** - The language excludes pointers to retain feasible verifiability.
- **No aliasing** - At every point in the program each variable has a unique name.
- **No side effects** - Function subprograms are pure mathematical functions.
- **No dynamic memory allocation or recursion** - The language excludes dynamic memory allocation and recursion. Consequently, it is relatively straight forward to calculate the maximum memory usage of a program.
- **Single threaded** - The language is single threaded, avoiding the various complexities associated with concurrent programs. Note that an extension to SPARK, called RavenSPARK, supports multiple program threads.

4.3.2 Relationship to Ada

The SPARK programming language is expressed as a subset of Ada. This is simply a practical manoeuvre to avoid implementing and maintaining a SPARK compiler on numerous architectures. Thus, SPARK should be regarded as a separate programming language, that just happens to be expressed as a subset of Ada. Nevertheless, this relationship restricts the application of the SPARK Approach to architectures that have an Ada compiler.

In practice, this is not a concern, as architectures associated with high integrity software typically have robust Ada compiler support.

The SPARK language is divided into a common kernel and annotations, as illustrated in Figure 4.1. The common kernel is expressed through a subset of Ada. The annotations are embedded inside Ada comments and expressed through notations specific to SPARK. As the annotations are inside Ada comments they remain legal Ada syntax. There are three different Ada standards as Ada 83 [Ame83], Ada 95 [Int95] and Ada 2005 [Int07]. Associated with each Ada standard is a corresponding subset of SPARK. The differences between these SPARK subsets is very marginal. Throughout this thesis we focus on the SPARK subset that corresponds to Ada 95.

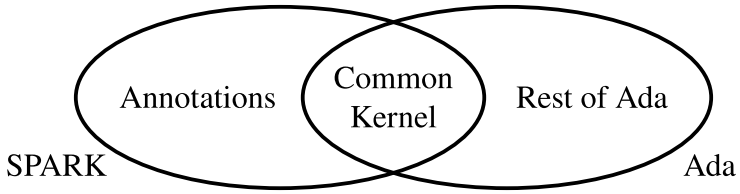


Figure 4.1: SPARK and Ada

4.3.3 Example

Consider the FilterInteger subprogram shown in Figure 4.2. The subprogram sums all of the elements in an array that lie between 0 and 100. The subprogram is contained within a package called FilterInteger_Package. The package is split into a *package specification* (ADS) file and a *package body* (ADB) file. The specification serves as a contract, describing *what* functionality is provided. The body implements the contract, describing *how* the functionality is achieved. The specification of the subprogram includes a dependency relation, introduced through derives annotation (`--# derives`). This specifies the information flow of the subprogram, as explained in §4.4.3. The body of the subprogram includes an invariant, introduced through the assert annotation (`--# assert`). This specifies a property that remains true within the loop, as explained in §4.4.4.

4.4 The SPARK toolset

The SPARK toolset supports the analysis of programs written in SPARK. Each of the tools are briefly summarised below:

- **SPARK Examiner** (henceforth Examiner) - A static analysis tool, supporting various analyses of SPARK programs.
- **SPADE Simplifier** (henceforth Simplifier) - A human-oriented automated theorem prover and simplifier, applied during program verification.

Package Specification (ADS)

```
package FilterInteger_Package is
  subtype AR_T is Integer range 0..9;
  type A_T is array (AR_T) of Integer;
  procedure FilterInteger(A: in A_T; R: out Integer);
  --# derives R from A;
end FilterInteger_Package;
```

Package Body (ADB)

```
package body FilterInteger_Package is
  procedure FilterInteger(A: in A_T; R: out Integer)
  is
  begin
    R:=0;
    for I in AR_T loop
      --# assert R>=0 and R<=I*100;
      if A(I)>=0 and A(I)<=100 then
        R:=R+A(I);
      end if;
    end loop;
  end FilterInteger;
end FilterInteger_Package;
```

Figure 4.2: FilterInteger subprogram

- **SPADE Proof Checker** (henceforth Checker) - An interactive proof assistant, applied during program verification.
- **Proof Obligation Summary Tool** (henceforth POGS) - A report generator, describing the current status of a program verification.

The interaction of these tools is illustrated in Figure 4.3. The main activities supported by the SPARK toolset are described in the sections below.

4.4.1 Conformance to SPARK

The Examiner checks that submitted source code conforms to the SPARK language. The check is mandatory, as all of the favourable properties of the SPARK Approach depend on reasoning about well-formed SPARK programs. Any conformance errors are highlighted in the *Examiner report* (REP) file.

4.4.2 Data Flow Analysis

The Examiner supports automated data flow analysis. The analysis is mandatory as a data flow error could undermine the security of a SPARK program. The data flow analysis checks that the parameters and global variables accessed in a subprogram are used according to their declared modes. Further, it is checked that all variables are written to before being read. Finally, any structurally inaccessible code is identified. Any data flow errors are highlighted in the *Examiner report* (REP) file.

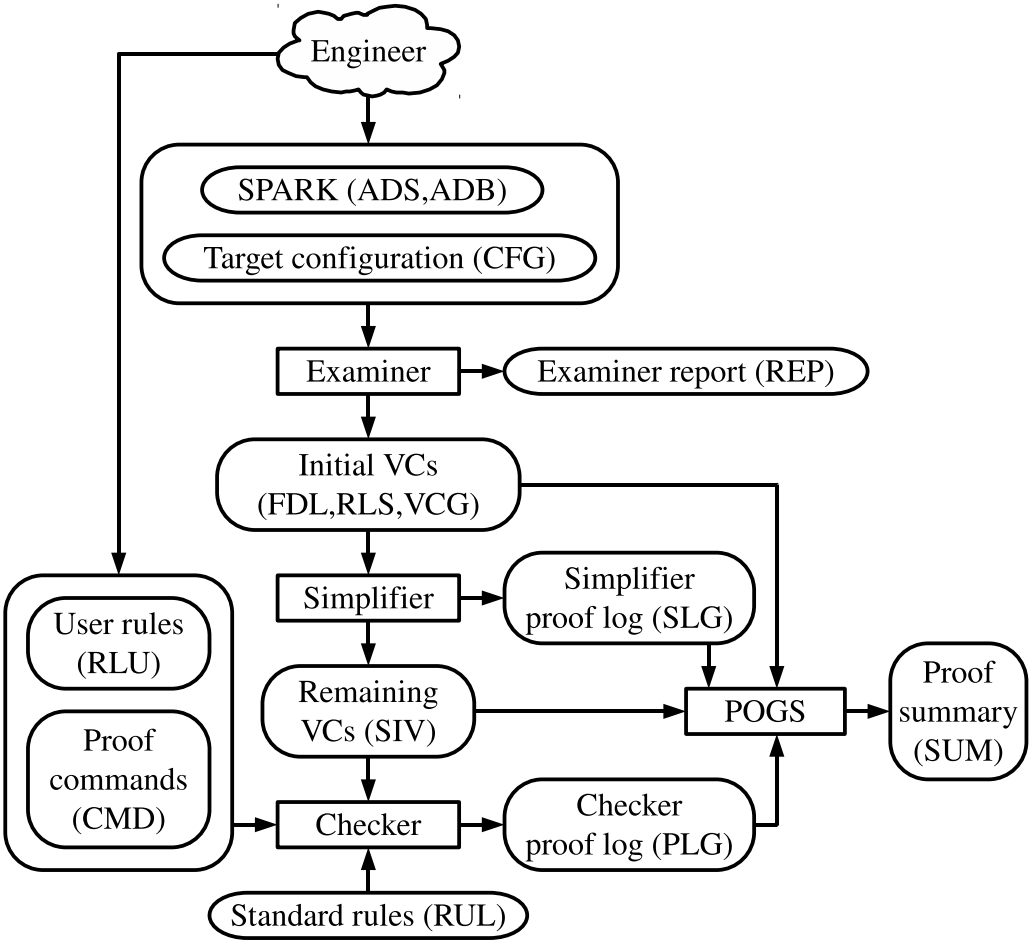


Figure 4.3: The SPARK toolset

4.4.3 Information Flow Analysis

The Examiner supports automated information flow analysis. Although information flow analysis is optional, it is applied in many projects. Information flow analysis compares specified information flow against actual information flow. Any discrepancies are highlighted in the *Examiner report* (REP) file. The analysis is effective in detecting both common and surprising program errors. The specification is provided in the form of *dependency relations* through *derives annotations*. A dependency relation lists, for each output from a subprogram, every input value that the output *may* depend on.

For example, consider the Switch subprogram shown in Figure 4.4. This simple subprogram swaps the values of two input parameters. The derives annotation attached to the specification of the subprogram is:

```
--# derives X from Y &
--#         Y from X;
```

This indicates that the output value of variable *x* will depend solely on the input value of variable *y* and that the output value of variable *y* will depend solely on the input value of variable *x*. In swapping the contents of variable *x* and *y* the subprogram correctly implements this specification.

```
package Switch_Package
is
  procedure Switch(X, Y: in out Integer);
  --# derives X from Y &
  --#         Y from X;
end Switch_Package;
```

```
package body Switch_Package is
  procedure Switch(X, Y: in out Integer)
  is
    T: Integer;
  begin
    T:=X; X:=Y; Y:=T;
  end Switch;
end Switch_Package;
```

Figure 4.4: Switch subprogram

4.4.4 Program Verification

The SPARK Approach supports program verification through axiomatic assertional reasoning. Although program verification is optional, it is typically applied to some extent in high integrity software development. The two properties that may be verified of SPARK programs are *partial correctness* and *exception freedom*. The specific details of verifying these properties is described in §4.4.5 and §4.4.6. The general process of program verification is described below.

In the SPARK Approach, program verification is decompositional. The whole program is verified by separately verifying each subprogram. Subprogram specifications are conveyed through annotations as *proof assertions*. The specification of each subprogram is described through a *precondition* and a *postcondition*. Properties that hold within loops are described through an *invariant*. These annotations may be expressed in terms of declared *proof functions*. Definitions and proprieties may be provided in external *user rule* (RLU) files. Further, target specific constraints, such as the size of base types, may be specified through a *target configuration* (CFG) file.

The Examiner operates as a verification condition generator, receiving a SPARK program and generating VCs for each subprogram. Unless explicitly specified, every subprogram is assigned a default precondition and postcondition of *true*. Similarly, unless already present, each loop is assigned a default invariant. Every invariant is strengthened to assert that imported parameter variables are within their type and that the precondition was true on entry to the subprogram. The VCs associated with each subprogram are expressed through three files. Firstly, a *functional description language* (FDL) file describes the entities and types that are relevant to the subprogram. Secondly, a *subprogram rules* file (RLS) contains a collection of rules that are specific to the entities and types in the subprogram. Thirdly, a *verification condition* (VCG) file contains the actual VCs. The VCs are described in first order logic with equality.

Once generated, VCs are presented to the Simplifier, seeking automated proof or simplification. The Simplifier generates additional files to describe its analysis. All remaining VCs are stored in a *simplified verification condition* (SIV) file. Further, the actions taken by the Simplifier are recorded in a *Simplifier proof log* (SLG) file. The user may interactively prove remaining VCs via the Checker. To facilitate reasoning, the Checker is supplied with a collection of general definitions and proprieties in external *standard rule* (RUL) files. Each Checker session is stored in a *proof command* (CMD) file, providing an audit trail and allowing for a proof effort to be automatically repeated. The progress of an interactive proof is recorded in a *Checker proof log* (PLG) file.

The status of a program verification is found by combining the status of every subprogram verification. Each subprogram verification depends on the VCG files generated by the Examiner, the SIV and SLG files generated by the Simplifier and the PLG file generated by the Checker. POGS collates these information sources, generating a *proof summary* (SUM) file.

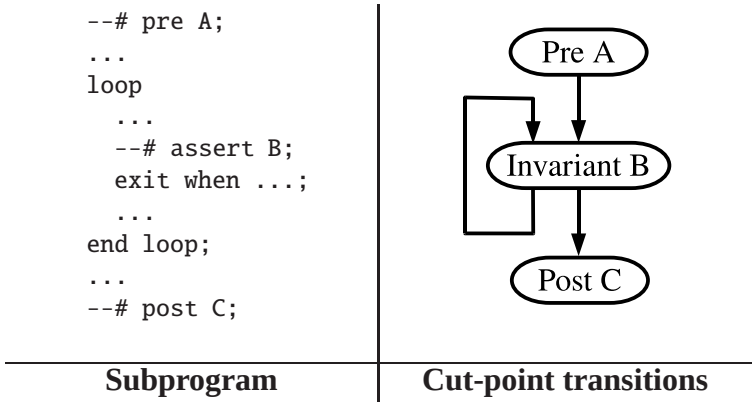
Aside from user rules, the soundness of a program verification depends entirely on the soundness of the SPARK Approach. In particular, the soundness of a program verification depends on the soundness of the Examiner, Simplifier and Checker.

4.4.5 Partial Correctness

Partial correctness verifies, for each subprogram, that where its precondition holds and the subprogram terminates, its postcondition will hold. The effort required in verifying

partial correctness depends on the richness of the subprogram specification. In general, as the specification becomes stronger, increased manual proof and invariant discovery is required. Consequently, verification of partial correctness is typically targeted at critical specifications and critical areas of functionality. For example, the safety critical core of the SHOLIS system was subjected to a targeted proof of partial correctness [Cha00].

As described in §4.4.4, program verification involves generating and proving VCs for each subprogram. Each proof assertion represents a *cut-point* in the subprogram. Proving every transition between cut-points proves that the overall subprogram is correct. A VC is generated for each path between cut-points. In verifying partial correctness, the relationship between cut-points and VCs is illustrated in Figure 4.5.



On the left, a generic subprogram containing a single loop is show, associating labels with each cut-point. On the right, cut-point transitions corresponding to the subprogram are shown. Each bold arrow represents every potential path between cut-points. A VC is generated for each of these paths.

Figure 4.5: Generating partial correctness VCs

For illustration, a small example is considered, showing the main artifacts of verifying partial correctness in the SPARK Approach. Consider the PolishFlag subprogram shown in Figure 4.6. The subprogram sorts an array of coloured elements. The subprogram has been specified through a precondition (`--# pre`) and a postcondition (`--# post`). The precondition specifies that the array contains only white and red elements. The postcondition states that the array contains a reordering of the input elements, with the first portion of the array containing white elements and the second portion containing red elements[1]. An invariant (`--# assert`) describes the partially sorted array. The package specification declares *permutation* as a user proof function (`--# function`). The proof function is used to specify that the output array is a reordering of the input array. The function is defined in a user rule file as shown in Figure 4.7. The logical interpretation of such rule files is described in §6.6.1. Target specific constraints are described through

[1]Thus depicting the vertical display of the Polish flag as specified in the "Coat of Arms, Colors and Anthem of the Republic of Poland, and State Seals Act" of 1980.

```
package PolishFlag_Package is
  subtype FlagIndex is Integer range 1..10;
  type Colour is (White, Red);
  type FlagArray is array (FlagIndex) of Colour;
  --# function Permutation(A: FlagArray ; B: FlagArray) return Boolean;
  procedure PolishFlag(Flag: in out FlagArray);
  --# derives Flag from Flag;
  --# pre (for all I in FlagIndex =>
  --#                   (Flag(I)=White or Flag(I)=Red));
  --# post
  --# (for some P in Integer range (Flag'First)..(Flag'Last+1) =>
  --#                ((for all Q in Integer range Flag'First..(P-1) =>
  --#                                (Flag(Q)=White)) and
  --#                 (for all R in Integer range P..Flag'Last =>
  --#                                (Flag(R)=Red)))) and
  --# Permutation(Flag, Flag~);
end PolishFlag_Package;
```

```
package body PolishFlag_Package is
  procedure PolishFlag(Flag: in out FlagArray)
  is
  subtype FlagIndexPlus is Integer range Flag'First..Flag'Last+1;
  I: FlagIndexPlus;
  J: FlagIndexPlus;
  T: Colour;
  begin
    I:=FlagIndexPlus'First;
    J:=FlagIndexPlus'Last;
    loop
      --# assert
      --# (for all Q in Integer range Flag'First..(I-1) =>
      --#               (Flag(Q)=White)) and
      --# (for all R in Integer range J..Flag'Last =>
      --#               (Flag(R)=Red)) and
      --# I in FlagIndexPlus and
      --# J in FlagIndexPlus and
      --# Permutation(Flag, Flag~);
      exit when not (I<J);
      if Flag(I)=White then
        I:=I+1;
      else
        J:=J-1;
        T:=Flag(I);
        Flag(I):=Flag(J);
        Flag(J):=T;
      end if;
    end loop;
  end PolishFlag;
end PolishFlag_Package;
```

Figure 4.6: PolishFlag subprogram (partial correctness)

a configuration file, as shown in Figure 4.8. The configuration file describes the range of integer types on a standard 32-bit architecture. Note that, unless stated otherwise, every non-industrial example in this thesis adopts this configuration file.

```
rule_family permutation:
    permutation(X, Y) requires [X: any, Y: any].

permutation(1): permutation(A, A) may_be_deduced.

permutation(2): permutation(A, B) may_be_replaced_by permutation(B, A).

permutation(3): permutation(update(update(A, [I], X), [J], Y), B)
                may_be_replaced_by
                permutation(update(update(A, [J], X), [I], Y), B).
```

Figure 4.7: Definition of Permutation (RLU)

```
package Standard is
    type Short_Short_Integer is range -2**7 .. 2**7-1;
    type Short_Integer is range -2**15 .. 2**15-1;
    type Integer is range -2**31 .. 2**31-1;
    type Long_Integer is range -2**31 .. 2**31-1;
    type Long_Long_Integer is range -2**63 .. 2**63-1;
end Standard;
```

Figure 4.8: Target Configuration (CFG)

As described in §4.4.4, the VCs corresponding to a subprogram are expressed through three files. The FDL file, shown in Figure 4.9, declares those entities and types that are relevant to the subprogram. The RLS file, shown in Figure 4.10, contains rules directly related to the entities seen in the subprogram. Both the FDL and RLS files are omitted in all subsequent examples, as they can be intuitively inferred from the subprogram source code. The VCG file is split across Figure 4.11 and Figure 4.12. A *traceability line* is shown for every pair of traversable cut-points. Each traceability line is followed by its corresponding VCs, one for each path between these cut-points. Four partial correctness VCs are present. One VC is generated from the precondition to the invariant. Two VCs are generated from the invariant to the invariant, covering both paths through the if-statement. Finally, one VC is generated from the invariant to the postcondition.

The initial VCs are presented to the Simplifier, generating a SIV file as shown in Figure 4.13. In general, the Simplifier offers limited automation in proving partial correctness VCs. In this case, only simple inequality conclusions are automatically proved.

The remaining VCs may be proved in an interactive Checker session. A CMD file that proves every remaining VC is shown in Appendix A. Typically, creating such files is a non-trivial task. The interactive application of the Checker lies beyond the focus of this thesis. Full details of the Checker and its proof commands are available in [Prab].

```
title procedure polishflag;

  function round__(real) : integer;
  type colour = (white, red);
  type flagarray = array [integer] of colour;
  const flagindexplus__last : integer = pending;
  const flagindexplus__first : integer = pending;
  const colour__last : colour = pending;
  const colour__first : colour = pending;
  const flagindex__last : integer = pending;
  const flagindex__first : integer = pending;
  const integer__last : integer = pending;
  const integer__first : integer = pending;
  var j : integer;
  var i : integer;
  var flag : flagarray;
  function permutation(flagarray, flagarray) : boolean;

end;
```

Figure 4.9: PolishFlag subprogram declarations (FDL)

```
rule_family polishflag_rules:
     X      requires [X:any] &
     X <= Y requires [X:ire, Y:ire] &
     X >= Y requires [X:ire, Y:ire].

polishflag_rules(1): character__pos(X) may_be_replaced_by X.
polishflag_rules(2): character__val(X) may_be_replaced_by X.
polishflag_rules(3): integer__first may_be_replaced_by -2147483648.
polishflag_rules(4): integer__last may_be_replaced_by 2147483647.
polishflag_rules(5): integer__base__first may_be_replaced_by -2147483648.
polishflag_rules(6): integer__base__last may_be_replaced_by 2147483647.
polishflag_rules(7): flagindex__first may_be_replaced_by 1.
polishflag_rules(8): flagindex__last may_be_replaced_by 10.
polishflag_rules(9): flagindex__base__first may_be_replaced_by -2147483648.
polishflag_rules(10): flagindex__base__last may_be_replaced_by 2147483647.
polishflag_rules(11): colour__first may_be_replaced_by white.
polishflag_rules(12): colour__last may_be_replaced_by red.
polishflag_rules(13): colour__base__first may_be_replaced_by white.
polishflag_rules(14): colour__base__last may_be_replaced_by red.
polishflag_rules(15): colour__pos(colour__first) may_be_replaced_by 0.
polishflag_rules(16): colour__pos(white) may_be_replaced_by 0.
polishflag_rules(17): colour__val(0) may_be_replaced_by white.
polishflag_rules(18): colour__pos(red) may_be_replaced_by 1.
polishflag_rules(19): colour__val(1) may_be_replaced_by red.
polishflag_rules(20): colour__pos(colour__last) may_be_replaced_by 1.
polishflag_rules(21): colour__pos(succ(X)) may_be_replaced_by
     colour__pos(X) + 1
     if [X <=red, X <> red].
polishflag_rules(22): colour__pos(pred(X)) may_be_replaced_by
     colour__pos(X) - 1
     if [X >=white, X <> white].
polishflag_rules(23): colour__pos(X) >= 0 may_be_deduced_from
     [white <= X, X <= red].
polishflag_rules(24): colour__pos(X) <= 1 may_be_deduced_from
     [white <= X, X <= red].
polishflag_rules(25): colour__val(X) >= white may_be_deduced_from
     [0 <= X, X <= 1].
polishflag_rules(26): colour__val(X) <= red may_be_deduced_from
     [0 <= X, X <= 1].
polishflag_rules(27): succ(colour__val(X)) may_be_replaced_by
     colour__val(X+1)
     if [0 <= X, X < 1].
polishflag_rules(28): pred(colour__val(X)) may_be_replaced_by
     colour__val(X-1)
     if [0 < X, X <= 1].
polishflag_rules(29): colour__pos(colour__val(X)) may_be_replaced_by X
     if [0 <= X, X <= 1].
polishflag_rules(30): colour__val(colour__pos(X)) may_be_replaced_by X
     if [white <= X, X <= red].
polishflag_rules(31): colour__pos(X) <= colour__pos(Y) & X <= Y are_interchangeable
     if [white <= X, X <= red, white <= Y, Y <= red].
polishflag_rules(32): colour__val(X) <= colour__val(Y) & X <= Y are_interchangeable
     if [0 <= X, X <= 1, 0 <= Y, Y <= 1].
polishflag_rules(33): flagindexplus__first may_be_replaced_by 1.
polishflag_rules(34): flagindexplus__last may_be_replaced_by 11.
polishflag_rules(35): flagindexplus__base__first may_be_replaced_by -2147483648.
polishflag_rules(36): flagindexplus__base__last may_be_replaced_by 2147483647.
```

Figure 4.10: PolishFlag subprogram rules (RLS)

```
For path(s) from start to assertion of line 13:

procedure_polishflag_1.
H1:     for_all(i_: integer, ((i_ >= flagindex__first) and (
           i_ <= flagindex__last)) -> ((element(flag, [i_]) =
           white) or (element(flag, [i_]) = red))) .
H2:     for_all(i___1: integer, ((i___1 >= flagindex__first) and (
           i___1 <= flagindex__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
        ->
C1:     for_all(q_: integer, ((q_ >= flagindex__first) and (
           q_ <= flagindexplus__first - 1)) -> (element(
           flag, [q_]) = white)) .
C2:     for_all(r_: integer, ((r_ >= flagindexplus__last) and (
           r_ <= flagindex__last)) -> (element(flag, [r_]) =
           red)) .
C3:    flagindexplus__first >= flagindexplus__first .
C4:    flagindexplus__first <= flagindexplus__last .
C5:    flagindexplus__last >= flagindexplus__first .
C6:    flagindexplus__last <= flagindexplus__last .
C7:    permutation(flag, flag) .


For path(s) from assertion of line 13 to assertion of line 13:

procedure_polishflag_2.
H1:     for_all(q_: integer, ((q_ >= flagindex__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = white)) .
H2:     for_all(r_: integer, ((r_ >= j) and (r_ <=
           flagindex__last)) -> (element(flag, [r_]) = red)) .
H3:    i >= flagindexplus__first .
H4:    i <= flagindexplus__last .
H5:    j >= flagindexplus__first .
H6:    j <= flagindexplus__last .
H7:    permutation(flag, flag~) .
H8:    not (not (i < j)) .
H9:    element(flag, [i]) = white .
        ->
C1:     for_all(q_: integer, ((q_ >= flagindex__first) and (
           q_ <= i + 1 - 1)) -> (element(flag, [q_]) =
           white)) .
C2:     for_all(r_: integer, ((r_ >= j) and (r_ <=
           flagindex__last)) -> (element(flag, [r_]) = red)) .
C3:    i + 1 >= flagindexplus__first .
C4:    i + 1 <= flagindexplus__last .
C5:    j >= flagindexplus__first .
C6:    j <= flagindexplus__last .
C7:    permutation(flag, flag~) .
```

Figure 4.11: PolishFlag subprogram VCs (VCG) [1 of 2]

```
procedure_polishflag_3.
H1:     for_all(q_: integer, ((q_ >= flagindex__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = white)) .
H2:     for_all(r_: integer, ((r_ >= j) and (r_ <=
           flagindex__last)) -> (element(flag, [r_]) = red)) .
H3:    i >= flagindexplus__first .
H4:    i <= flagindexplus__last .
H5:    j >= flagindexplus__first .
H6:    j <= flagindexplus__last .
H7:    permutation(flag, flag~) .
H8:    not (not (i < j)) .
H9:    not (element(flag, [i]) = white) .
        ->
C1:     for_all(q_: integer, ((q_ >= flagindex__first) and (
           q_ <= i - 1)) -> (element(update(update(flag, [i], element(
           flag, [j - 1])), [j - 1], element(flag, [i])), [
           q_]) = white)) .
C2:     for_all(r_: integer, ((r_ >= j - 1) and (r_ <=
           flagindex__last)) -> (element(update(update(flag, [
           i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [r_]) = red)) .
C3:    i >= flagindexplus__first .
C4:    i <= flagindexplus__last .
C5:    j - 1 >= flagindexplus__first .
C6:    j - 1 <= flagindexplus__last .
C7:    permutation(update(update(flag, [i], element(flag, [
           j - 1])), [j - 1], element(flag, [i])), flag~) .


For path(s) from assertion of line 13 to finish:

procedure_polishflag_4.
H1:     for_all(q_: integer, ((q_ >= flagindex__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = white)) .
H2:     for_all(r_: integer, ((r_ >= j) and (r_ <=
           flagindex__last)) -> (element(flag, [r_]) = red)) .
H3:    i >= flagindexplus__first .
H4:    i <= flagindexplus__last .
H5:    j >= flagindexplus__first .
H6:    j <= flagindexplus__last .
H7:    permutation(flag, flag~) .
H8:    not (i < j) .
        ->
C1:     for_some(p_: integer, ((p_ >= flagindex__first) and (
           p_ <= flagindex__last + 1)) and (( for_all(q_:
           integer, ((q_ >= flagindex__first) and (q_ <= p_ - 1)) -> (element(
           flag, [q_]) = white))) and ( for_all(r_:
           integer, ((r_ >= p_) and (r_ <= flagindex__last)) -> (element(
           flag, [r_]) = red))))) .
C2:    permutation(flag, flag~) .
```

Figure 4.12: PolishFlag subprogram VCs (VCG) [2 of 2]

```
For path(s) from start to assertion of line 13:

procedure_polishflag_1.
H1:    for_all(i_ : integer, 1 <= i_ and i_ <= 10 -> element(flag, [i_]) =
          white or element(flag, [i_]) = red) .
H2:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 10 -> white <= element(
          flag, [i___1]) and element(flag, [i___1]) <= red) .
       ->
C1:    for_all(q_ : integer, 1 <= q_ and q_ <= 0 -> element(flag, [q_]) = white) .
C2:    for_all(r_ : integer, 11 <= r_ and r_ <= 10 -> element(flag, [r_]) = red) .
C7:    permutation(flag, flag) .


For path(s) from assertion of line 13 to assertion of line 13:

procedure_polishflag_2.
H1:    for_all(q_ : integer, 1 <= q_ and q_ <= i - 1 -> element(flag, [q_]) =
          white) .
H2:    for_all(r_ : integer, j <= r_ and r_ <= 10 -> element(flag, [r_]) = red) .
H3:    i >= 1 .
H4:    j <= 11 .
H5:    permutation(flag, flag~) .
H6:    i < j .
H7:    element(flag, [i]) = white .
       ->
C1:    for_all(q_ : integer, 1 <= q_ and q_ <= i -> element(flag, [q_]) = white) .


procedure_polishflag_3.
H1:    for_all(q_ : integer, 1 <= q_ and q_ <= i - 1 -> element(flag, [q_]) =
          white) .
H2:    for_all(r_ : integer, j <= r_ and r_ <= 10 -> element(flag, [r_]) = red) .
H3:    i >= 1 .
H4:    j <= 11 .
H5:    permutation(flag, flag~) .
H6:    i < j .
H7:    element(flag, [i]) <> white .
       ->
C1:    for_all(q_ : integer, 1 <= q_ and q_ <= i - 1 -> element(update(update(
          flag, [i], element(flag, [j - 1])), [j - 1], element(flag, [i])), [q_]
          ) = white) .
C2:    for_all(r_ : integer, j - 1 <= r_ and r_ <= 10 -> element(update(update(
          flag, [i], element(flag, [j - 1])), [j - 1], element(flag, [i])), [r_]
          ) = red) .
C7:    permutation(update(update(flag, [i], element(flag, [j - 1])), [j - 1],
          element(flag, [i])), flag~) .


For path(s) from assertion of line 13 to finish:

procedure_polishflag_4.
H1:    for_all(q_ : integer, 1 <= q_ and q_ <= i - 1 -> element(flag, [q_]) =
          white) .
H2:    for_all(r_ : integer, j <= r_ and r_ <= 10 -> element(flag, [r_]) = red) .
H3:    i >= 1 .
H4:    i <= 11 .
H5:    j >= 1 .
H6:    j <= 11 .
H7:    permutation(flag, flag~) .
H8:    j <= i .
       ->
C1:    for_some(p_ : integer, p_ >= 1 and p_ <= 11 and (for_all(q_ : integer, 1
          <= q_ and q_ <= p_ - 1 -> element(flag, [q_]) = white) and for_all(r_
          : integer, p_ <= r_ and r_ <= 10 -> element(flag, [r_]) = red))) .
```

Figure 4.13: PolishFlag subprogram simplified VCs (SIV)

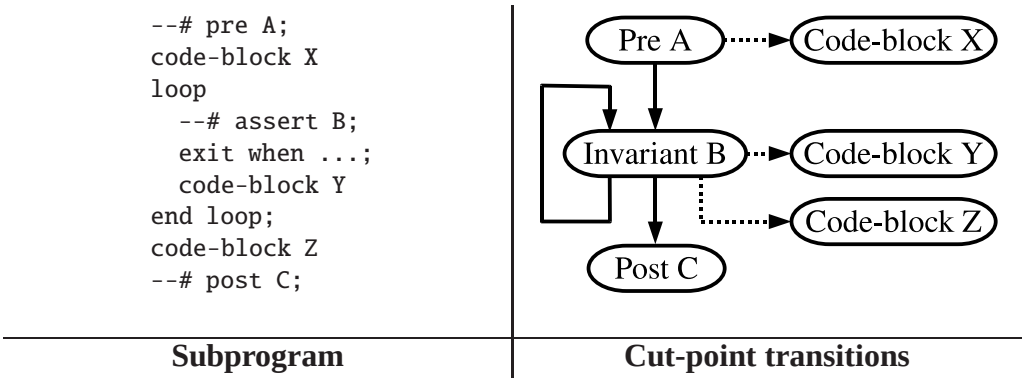
4.4.6 Exception Freedom

Exception freedom verifies, for each subprogram, that where its precondition is met and the subprogram terminates, its postcondition will hold and the subprogram will not raise an exception. As SPARK is a subset of Ada, it must be verified that no Ada exceptions can occur. The predefined Ada 95 exceptions [Int95] (11.1(4)) are considered below, describing how their absence is verified in SPARK:

- **Tasking_Error** - Exceptions of this category are raised where errors are detected during intertask communication. As SPARK is single threaded, tasking errors can never occur.
- **Storage_Error** - Exceptions of this category are raised where there is not sufficient memory to perform an operation. Such errors may occur in SPARK. However, due to the language requirements of SPARK, it is relatively straight forward to calculate the maximum memory usage of a program. By ensuring that this memory is available at run-time, storage errors can never occur.
- **Program_Error** - Exceptions of this category are raised for various program defects that can arise during execution. These defects either fall outside the SPARK subset or are detected automatically during static analysis. Thus, for well-formed SPARK, program errors can never occur.
- **Constraint_Error** - Exceptions of this category are raised where declared or architecture constraints are violated. Many of these defects fall outside the SPARK subset, and thus can never occur. However, four of the exceptions in this category can arise in SPARK. The absence of the following exceptions is demonstrated through program verification:
  - **Index_Check** - Checks that an array access occurs within the declared bounds of the array.
  - **Range_Check** - Checks that values remain within the declared bounds of their types.
  - **Division_Check** - Checks that the denominator of a division, remainder, or modulus operation is not zero.
  - **Overflow_Check** - Checks that a numeric operation does not overflow the working memory space.

Verifying exception freedom involves verifying that expressions remain within certain constraints. Sufficient constraints are often available by adopting a strong type model. Further, given the targeted verification task, significant proof automation is often achievable. Consequently, the verification of exception freedom is often undertaken for complete high integrity software systems. For example, the entire SHOLIS system was subjected to a proof of exception freedom [Cha00].

The process of verifying exception freedom is an extension of that seen where proving partial correctness, as illustrated in Figure 4.5. Additional *run-check* cut-points are automatically introduced, reflecting the exceptions that can occur in SPARK. In verifying exception freedom, the relationship between cut-points and VCs is illustrated in Figure 4.14.



On the left, a generic subprogram containing a single loop is show, associating labels with each cut-point and each code-block. On the right, the cut-point transitions corresponding to the subprogram are shown. Each bold arrow represents every potential path between cut-points. A VC is generated for each of these paths. Each dotted arrow represents every potential path from a cut-point to a statement that may raise an exception. A VC is generated for each of these paths.

Figure 4.14: Generating exception freedom VCs

For illustration, consider again the PolishFlag subprogram. The specification of the subprogram is weakened to verify only exception freedom, as shown in Figure 4.15. Note that, purely for clarity, an explicit invariant (`--# assert true;`) is retained.

In verifying exception freedom, twelve VCs are generated. Four of these VCs correspond to verifying partial correctness of the default specification. A VC is generated for lines 9 and 15 to prove that the value assigned to *i* is within type. A VC is generated for lines 10 and 17 to prove that the value assigned to *j* is within type. A VC is generated for line 14 to prove that *i* is a legal index of array *flag*. A VC is generated for line 18 to prove both that *i* is a legal index of array *flag* and that the value assigned to *t* is within type. A VC is generated for line 19 to prove both that variables *i* and *j* are legal indices of array *flag* and that the value assigned to the i^{th} element of array *flag* is within type. Finally, a VC is generated for line 20 to prove both that *j* is a legal index of array *flag* and that the value assigned to the j^{th} element of array *flag* is within type. For illustration, the VC corresponding to line 19 is shown in Figure 4.16. The Simplifier proves all of these VCs automatically, verifying that the subprogram is free from exceptions.

```
package PolishFlag_Package is
  subtype FlagIndex is Integer range 1..10;
  type Colour is (White, Red);
  type FlagArray is array (FlagIndex) of Colour;
  procedure PolishFlag(Flag: in out FlagArray);
  --# derives Flag from Flag;
end PolishFlag_Package;
```

```
 1  package body PolishFlag_Package is
 2    procedure PolishFlag(Flag: in out FlagArray)
 3    is
 4    subtype FlagIndexPlus is Integer range Flag'First..Flag'Last+1;
 5    I: FlagIndexPlus;
 6    J: FlagIndexPlus;
 7    T: Colour;
 8    begin
 9      I:=FlagIndexPlus'First;
10      J:=FlagIndexPlus'Last;
11      loop
12        --# assert true;
13        exit when not (I<J);
14        if Flag(I)=White then
15          I:=I+1;
16        else
17          J:=J-1;
18          T:=Flag(I);
19          Flag(I):=Flag(J);
20          Flag(J):=T;
21        end if;
22      end loop;
23    end PolishFlag;
24  end PolishFlag_Package;
```

Figure 4.15: PolishFlag subprogram (exception freedom)

```
For path(s) from assertion of line 12 to run-time check associated with
statement of line 19:

procedure_polishflag_10.
H1:    true .
H2:     for_all(i___1: integer, ((i___1 >= flagindex__first) and (
           i___1 <= flagindex__last)) -> ((element(flag, [
           i___1]) >= colour__first) and (element(flag, [
           i___1]) <= colour__last))) .
H3:    i >= flagindexplus__first .
H4:    i <= flagindexplus__last .
H5:    j >= flagindexplus__first .
H6:    j <= flagindexplus__last .
H7:    not (not (i < j)) .
H8:    i >= flagindexplus__first .
H9:    i <= flagindexplus__last .
H10:   i >= flagindex__first .
H11:   i <= flagindex__last .
H12:   not (element(flag, [i]) = white) .
H13:   j >= flagindexplus__first .
H14:   j <= flagindexplus__last .
H15:   j - 1 >= flagindexplus__first .
H16:   j - 1 <= flagindexplus__last .
H17:   i >= flagindexplus__first .
H18:   i <= flagindexplus__last .
H19:   element(flag, [i]) >= colour__first .
H20:   element(flag, [i]) <= colour__last .
H21:   i >= flagindex__first .
H22:   i <= flagindex__last .
H23:   j - 1 >= flagindexplus__first .
H24:   j - 1 <= flagindexplus__last .
        ->
C1:    element(flag, [j - 1]) >= colour__first .
C2:    element(flag, [j - 1]) <= colour__last .
C3:    j - 1 >= flagindex__first .
C4:    j - 1 <= flagindex__last .
C5:    i >= flagindex__first .
C6:    i <= flagindex__last .
```

Figure 4.16: PolishFlag exception freedom VC (line 19 of ADB in §4.15)

4.5 Configuring the SPARK toolset

Changes to the SPARK toolset have the potential to influence the low level detail of our approach. For this reason, we use a particular version of each component of the toolset. Further, the application of our approach requires features not readily supported by the standard toolset. The required features are introduced by modifying the toolset accordingly. Each component of the toolset is listed below, noting the version we use and any additional features that were introduced:

- **Examiner** - We use Examiner version 7.1d01 (January 2004). This is an internal version, not associated with a particular toolset release. For reference, this version lies between toolset release 7.0 (July 2003) and toolset release 7.2 (December 2004). A very minor change has been made to this Examiner so that its output is easier to process, as described in §B.2.

- **Simplifier** - We use Simplifier version 2.18 (March 2005). This is an internal version, not associated with a particular toolset release. For reference, version 2.17 was included in toolset release 7.2 (December 2004) and version 2.22 was included in toolset release 7.3 (April 2006). The actual Simplifier we use has been slightly modified to support the automated comparison of initial and remaining VCs, as described in §B.3.

- **Checker** - We use Checker version 2.03, included in toolset release 7.2 (December 2004). The actual Checker we use has been modified to support the automated proof of VCs, as detailed in §B.4. Note that the Checker is used to check the soundness of discovered proof plans. Thus, in a critical environment, these modifications would be subject to rigorous verification and validation. The implications of this concern are explored in §9.2.2.

Chapter 5

Enhancing the SPARK Approach with SPADEase

This chapter describes the practicalities of verifying exception freedom in the SPARK Approach. The process is described in §5.1, highlighting the main challenges in §5.2. We enhance this process through SPADEase, as described in §5.3, addressing the main challenges as described in §5.4.

5.1 Verifying Exception Freedom

The process of verifying exception freedom in the SPARK Approach is illustrated in Figure 5.1. The verification effort is decompositional, verifying the whole program by separately verifying each subprogram. Further, the verification effort is iterative, incrementally resolving defects until the verification is complete.

Each iteration beings with the Examiner generating initial VCs for each subprogram. The Simplifier attempts to automatically prove the initial VCs, storing those it fails to prove as remaining VCs. Where there are no remaining VCs, the verification is complete. Otherwise, an engineer must manually intervene to resolve the remaining VCs. The three classes of interactions that may be required are listed below:

- **Fix fault** - The VC is not provable as there is an inconsistency between the subprogram and its specification. The engineer must identify the source of the fault and fix the subprogram, its specification, or both.
- **Perform proof** - The VC is provable, but is not automatically proved by the Simplifier. The engineer must prove the VC inside the Checker[1].
- **Strengthen specification** - The VC is not provable as information necessary for

[1]The SPARK Approach also supports the creation of a *proof review* (PRV) file. Such files justify the correctness of a VC through an alternative process, such as independent review of an informal proof. In this thesis, we focus on mechanically checkable formal verification.

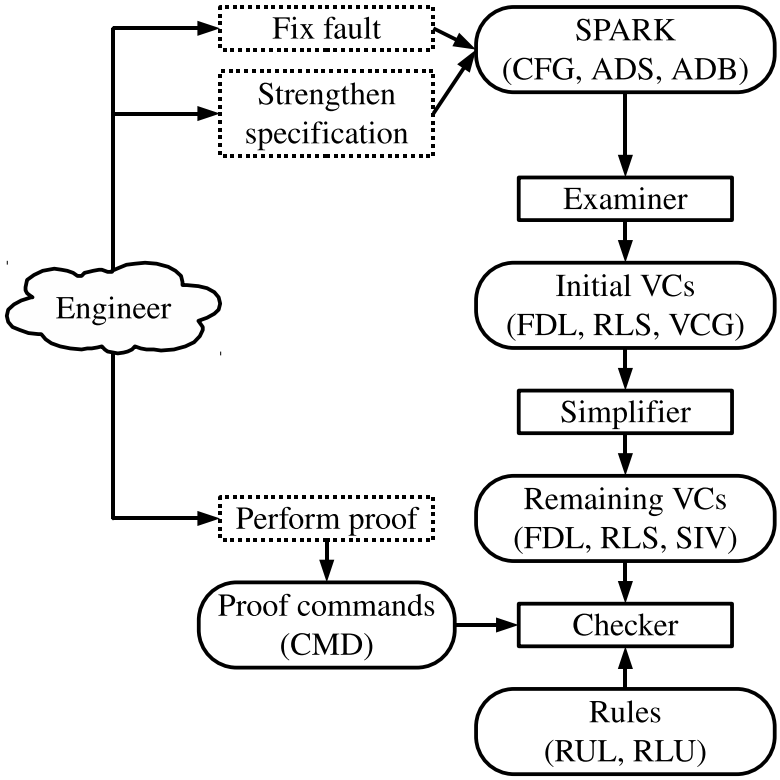


Figure 5.1: Verifying exception freedom

proof is missing from the specification. The engineer must strengthen the specification to introduce the missing information.

Where a subprogram or specification is modified they are no longer synchronised with the initial VCs. Thus, a further iteration of the verification process is required.

5.2 Verification Challenges

Industrial strength evidence [AC02] shows that the Simplifier can automatically prove around 90% of exception freedom VCs. More recent studies, with a later version of the Simplifier, suggest that this figure is now closer to 97% for engineered SPARK programs [JES07]. Despite these impressive results, verifying exception freedom can still require significant manual effort. The key challenges are listed below:

1. **Many remaining VCs** - A typical high integrity system will generate thousands of VCs. Thus, while the Simplifier may prove all but a small percentage of the initial VCs, the remaining VCs can still number in the hundreds.

2. **Complex remaining VCs** - The complexity of VCs is dependent on the complexity of the code constructs they reason about. Unsurprisingly, the Simplifier tends to be more effective at proving less complex VCs. Thus, the remaining VCs often reflect the more complex proof problems.

3. **Weak default invariants** - The automatically inserted default invariants are relatively weak. While the information is valuable, it is typically only sufficient to support the proof of simple loops.

4. **Lack of configuration** - The verification of a particular system may involve recurring patterns of both invariant discovery and proof discovery. However, the SPARK Approach can not be configured with appropriate strategies for these patterns. Instead, engineers must manually associate these patterns with their corresponding strategies and must manually perform these strategies.

5. **Brittle proofs** - The Checker is controlled through very specific proof commands. As a result, Checker proofs are tightly coupled to a particular VC and hence a particular instance of its corresponding subprogram. If the subprogram is modified it is very likely that the proof will need to be modified also.

5.3 Verifying Exception Freedom with SPADEase

The SPARK Approach is enhanced through SPADEase. The architecture of SPADEase is described in §5.3.1, while its application is described in §5.3.2.

5.3.1 Architecture of SPADEase

The process of verifying exception freedom in the enhanced SPARK Approach is illustrated in Figure 5.2. SPADEase strictly operates within the SPARK Approach, automating activities previously preformed manually by an engineer. Consequently, the soundness of the verification effort remains solely dependent on the soundness of the SPARK Approach.

The architecture of SPADEase is illustrated in Figure 5.3. The verification of exception freedom requires both proof discovery and invariant discovery. These distinct tasks are addressed through separate components. Proof discovery is achieved through a proof planner, while invariant discovery is achieved through a program analyser.

An objective for SPADEase is to deliver both tractable and sound configuration. Key features of proof planning naturally support this objective. Proof planning is controlled through external proof plans that reflect mathematical intuitions. Thus, the development of proof plans is relatively tractable. Further, proof planning makes a clear distinction between proof search and proof checking. Thus, any errors in proof plans will be detected during proof checking, and not undermine soundness. Recognising the value of these features, our program analyser is similarly configured. Program analysis is controlled through external program analysis heuristics. Further, the program analyser only discovers *candidate* invariants. Any errors in these invariants will be detected during the wider verification effort, and not undermine soundness.

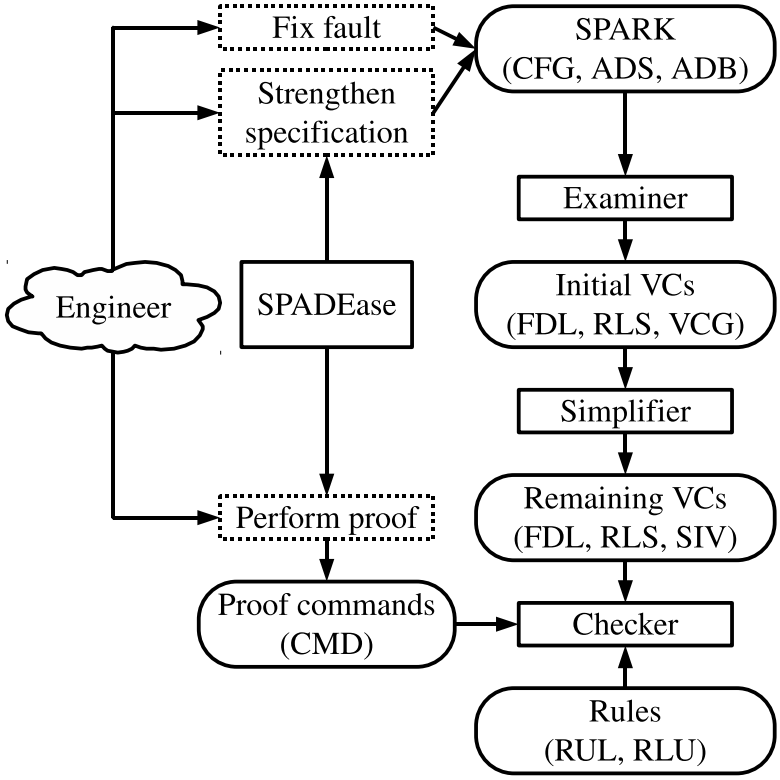


Figure 5.2: Verifying exception freedom with SPADEase

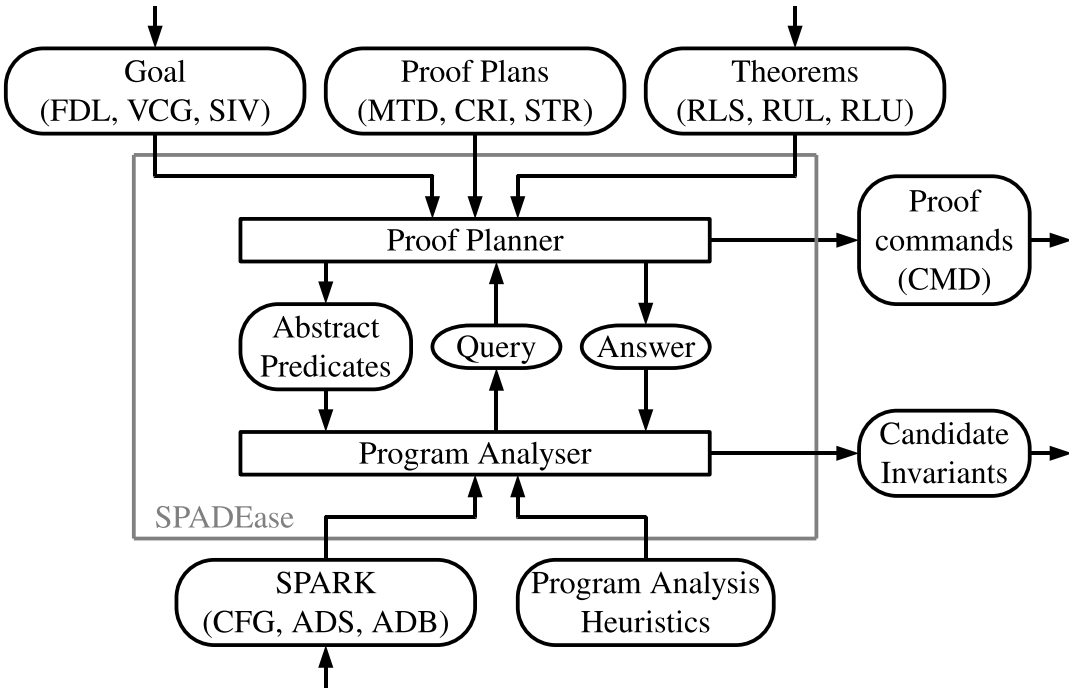


Figure 5.3: Architecture of SPADEase

A further objective for SPADEase is to deliver targeted invariant discovery through effective proof failure analysis. Proof planning directly supports proof failure analysis through its critics mechanism. However, a proof planner is not suited to performing program analysis. Thus, the critics mechanism is extended to communicate with a program analyser through *abstract predicates*. Similarly, the program analyser is not suited to reasoning about program properties. Thus, the program analyser communicates with the proof planner through a *query* and *answer* interface.

5.3.2 SPADEase Enhanced Verification Process

The verification process begins in exactly the same manner as the standard SPARK Approach, as described in §5.1. The verification effort is both decompositional and iterative. Each iteration begins with the Examiner generating initial VCs for each subprogram. The Simplifier attempts to prove the initial VCs, storing those that are not proved as remaining VCs. At this stage, each subprogram with remaining VCs is automatically investigated by SPADEase.

To begin, SPADEase investigates each remaining VC for a subprogram via the proof planner. Four scenarios may occur as detailed below:

- **Perform proof** - The proof planner successfully discovers a proof plan. The proof plan is extracted as proof commands and verified by the Checker.
- **Suggest specification strengthening** - The proof planner fails to discover a proof plan. However, failure analysis successfully identifies that missing information caused the failure. Specification strengthening is suggested, communicating the form of the missing information to the program analyser as an *abstract predicate*.
- **Suggest targeted interaction** - The proof planner fails to discover a proof plan. However, failure analysis successfully identifies the defect causing the failure. SPADEase is unable to automatically resolve the class of defect. Instead, the exact nature of the defect is communicated to an engineer, suggesting targeted interaction.
- **Fail** - Where none of the above scenarios occur, SPADEase is unable to advance the proof of the VC. However, analysis of other remaining VCs associated with the subprogram may suggest specification strengthening or targeted interaction. Resolving these related defects may indirectly advance the proof of this VC.

Once every remaining VC has been investigated, three scenarios may occur as detailed below:

- **Success** - Every remaining VC has been proved. SPADEase terminates successfully.

- **Strengthen specification** - If at least one abstract predicate was generated, the subprogram is subjected to program analysis. The program analyser attempts to discover invariants for the subprogram. During analysis, the proof planner is exploited to solve general reasoning queries. The program analyser does not verify the correctness of its discovered invariants. To reflect this, its discovered invariants are treated as *candidate invariants*. Those candidate invariants that satisfy the abstract predicates, and are not already present as a subprogram invariant, are selected to automatically strengthen the program specification. The correctness of every selected candidate invariant will be demonstrated during program verification.

- **Terminate, suggesting targeted interaction** - Where any targeted interaction is suggested, the application of SPADEase will terminate. The engineer must manually intervene, guided by the suggested interaction.

- **Terminate, in failure** - There are unproven remaining VCs, yet neither specification strengthening nor targeted interaction was suggested. In this situation, SPADEase terminates in failure. The engineer may choose to resolve the failure directly through the SPARK Approach. Alternatively, the engineer may choose to extend the heuristics of SPADEase such that the failure, and others of a similar pattern, will be automatically resolved in future.

Where the specification is modified, it is no longer synchronised with the initial VCs. Thus, a further iteration of the verification process is required. This may, in turn, trigger a subsequent application of SPADEase.

In general, the iterative process will terminate if SPADEase only suggests changes that advance the verification effort. Through a cooperative integration of the proof planner and program analyser, SPADEase operates in a strongly goal directed manner. Specification strengthening only introduces new invariants that address identified weaknesses in the VCs. Thus, SPADEase naturally makes genuine progress toward proof, and the iterative process is strongly expected to terminate.

5.4 Addressing Verification Challenges

The key challenges in verifying exception freedom in the SPARK Approach are listed in §5.2. SPADEase provides an effective infrastructure for addressing these challenges, as detailed below:

1. **Many remaining VCs** - SPADEase offers an additional layer of proof automation, potentially reducing the number of remaining VCs.

2. **Complex remaining VCs** - SPADEase performs automated proof discovery in a proof planner. As described in §3.3.1 and §3.3.4 proof planning delivers both ex-

tensibility and flexibility. Thus, SPADEase may be extended with sophisticated heuristics to address complex remaining VCs.

3. **Weak default invariants** - SPADEase performs automated invariant discovery in a program analyser. Appropriate heuristics may be developed to discover stronger invariants. Further, as invariants are introduced in reaction to proof failure, they are only introduced where necessary.

4. **Lack of configuration** - SPADEase makes a clear distinction between its infrastructure and its controlling heuristics. Further, by strictly automating the actions of an engineer, the SPARK Approach remains solely responsible for ensuring soundness. Thus, SPADEase is readily configurable without introducing any soundness concerns.

5. **Brittle proofs** - Rather than develop a specific proof inside the Checker, an engineer may choose to extend SPADEase with an appropriate heuristic. The heuristic should be expressed at a higher level of abstraction, and thus would not be tightly coupled to a particular VC. Further, the heuristic may be reused for all proofs that are susceptible to the same pattern. The development of general heuristics rather than specific proofs may not always be feasible. However, by actively adopting this technique where possible, the number of specific proofs developed in the Checker would be reduced.

Chapter 6

Proof Planner

6.1 Introduction

Following the proof planning paradigm, a proof planner is developed. The proof planner is tailored for an effective integration with the SPARK Approach. This chapter describes the details of our proof planner.

6.2 Proof Planner Architecture

The architecture of our proof planner is shown in Figure 6.1. The proof planner receives three distinct categories of input. The form of the *goal*, *theorems* and *proof plans* are discussed in §6.5, §6.6 and §6.7 respectively. These inputs are processed by the *proof planner*, as discussed in §6.8. The planner generates either an *instantiated proof plan* or a *failure critique*. Where an instantiated proof plan is generated, a *compound tactic* is extracted and checked within the Checker. Following this, an overall *result* is reported, as described in §6.9.

The architecture of our proof planner closely corresponds to the original proof planning architecture, as introduced in §3.2.2. The sole difference is that our proof planner may return a failure critique as the result. The mechanism is introduced to allow proof plans to critique on the wider verification effort.

6.3 Proof Planner Configuration

The proof planner is configured to support the verification of exception freedom in the SPARK Approach. A method-language is developed to support the expression of proof plans, as summarised in §6.10. The proof plans developed are summarised in §6.11.

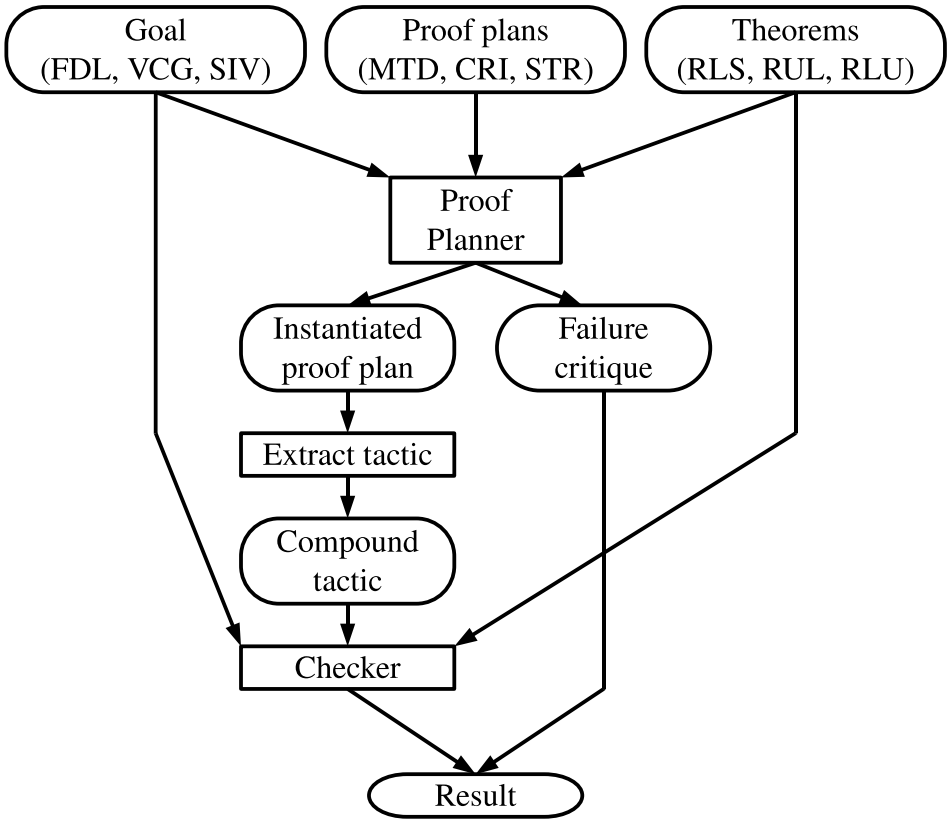


Figure 6.1: Proof planner architecture

6.4 Illustrative Example

The details of the proof planner are illustrated through a common example. Consider the SumOnAirTeletextPages subprogram shown in Figure 6.2. The enclosing package introduces data structures based on a Teletext system[1]. The subprogram totals the number of Teletext pages that are currently being broadcast. To support the verification of exception freedom, the for-loop has an invariant, constraining the total to be between zero and the number of pages inspected so far.

6.5 Goal

The proof planner operates in the same logic as the SPARK Approach, first order logic with equality. Mirroring the SPARK Approach, the description of a goal is retrieved in two distinct parts. Firstly, the entities and types relevant to goal are retrieved, as described in §6.5.1 and §6.5.2. Secondly, the structure of the goal itself is retrieved as described in §6.5.3.

[1] A 'Teletext' enabled television contains a decoder that converts data embedded into a television signal into pages of text and graphics. The Teletext system described here relates to the "Broadcast Teletext Specification, September 1976", which later formed the basis of the World System Teletext (WST) standard.

```
package SumOnAirTeletextPages_Package is
  --A teletext page is indexed: [MagazineDigit][PageDigitOne][PageDigitTwo]
  --For example: 888 (typically a subtitles page)
  subtype MagazineDigit is Integer range 1..8;
  subtype PageDigitOne is Integer range 0..9;
  subtype PageDigitTwo is Integer range 0..9;
  FirstPage: constant Integer:=
    ((MagazineDigit'First*100)+(PageDigitOne'First*10))+PageDigitTwo'First;
  LastPage:  constant Integer:=
    ((MagazineDigit'Last*100)+(PageDigitOne'Last*10))+PageDigitTwo'Last;
  subtype PagesIndex is Integer range FirstPage..LastPage;

  --Performance diminishes as more pages are broadcast.
  --Thus, typically, only a subset of the indexable pages are on-air.
  type PageStatus is (OnAir, OffAir);

  --A teletext page is made up from 24x40 blocks.
  subtype Rows is Integer range 0..23;
  subtype Columns is Integer range 0..39;
  type OneColumn is array (Columns) of Short_Short_Integer;
  type OneScreen is array (Rows) of OneColumn;
  type TeletextPage is record
    Status : PageStatus;
    Content : OneScreen;
  end record;
  type TeletextPages is array (PagesIndex) of TeletextPage;

  subtype Total is Integer range 0..((PagesIndex'Last-PagesIndex'First)+1);

  procedure SumOnAirTeletextPages(TP: in TeletextPages;
                                  R: out Total);
  --# derives R from TP;
end SumOnAirTeletextPages_Package;
```

```
1  package body SumOnAirTeletextPages_Package is
2    procedure SumOnAirTeletextPages(TP: in TeletextPages;
3                                    R: out Total)
4    is
5    begin
6      R:=0;
7      for I in PagesIndex loop
8        --# assert R>=0 and R<=(I-PagesIndex'First);
9        if (TP(I).Status=OnAir) then
10         R:=R+1;
11       end if;
12     end loop;
13   end SumOnAirTeletextPages;
14 end SumOnAirTeletextPages_Package;
```

Figure 6.2: SumOnAirTeletextPages subprogram

6.5.1 Declarations

All of the entities and types relevant to a goal are retrieved and stored for access during proof planning. Most of these entities and types are explicitly declared in the FDL file. However, some entities and types relevant to a goal are implicitly declared. For clarity, the proof planner does not have any implicit declarations. Note that, while the definitions described below are sufficient to express every VC considered in the thesis, they do not cover every VC of the SPARK Approach.

Scalar Types

A scalar type is not expressed in terms of any other type. Each scalar type describes an ordered set of values. The structure for holding scalar types is shown in Figure 6.3, and the types it holds are described below. For illustration, the corresponding structures for the SumOnAirTeletextPages subprogram are shown in Figure 6.4.

- **Real Numbers** - Mathematical real numbers of the set $\mathbb{R}$.
- **Integer Numbers** - Mathematical integer numbers of the set $\mathbb{Z}$.
- **Boolean Values** - Either the truth value *true* or *false*.
- **Enumerated Values** - Each enumerated type has a declared name *EnumRef* and a corresponding ordered list of identifiers <*EnumIdList*>.

Note that the SPARK Approach approximates fixed and floating point arithmetic using mathematical real numbers. The approximation ignores the rounding errors that occur for these machine types. Correctly and accurately reasoning about the impact of cumulative rounding errors is a challenging task, and an active area of research [Bar89, BF07]. Consequently, when reasoning about these types, particular care needs to be taken in interpreting verification results. To avoid such complexity, our techniques focus on discrete types.

<*Declaration*> ::= *scalarType*(<*Type*>) |
scalarEnumeratedType(*EnumRef*, <*EnumIdList*>)
<*Type*> ::= *real* | *integer* | *boolean*
<*EnumIdList*> ::= [] | [*EnumId* | <*EnumIdList*>]

Figure 6.3: Scalar types

Declarations from FDL
`type pagestatus = (onair, offair);`

Proof planner structures
scalarType(*real*)
scalarType(*integer*)
scalarType(*boolean*)
scalarEnumeratedType(*pagestatus*, [*onair*, *offair*])

Figure 6.4: SumOnAirTeletextPages scalar types

Composite Types

A composite type is expressed in terms of other types. A composite type may be composed from scalar types or other composite types. The structure for holding composite types is shown in Figure 6.5, and the types it holds are described below. For illustration, the corresponding structures for the SumOnAirTeletextPages subprogram are shown in Figure 6.6.

- **Arrays** - Each array type has a declared name *TypeRef*. An array contains a list of values, called elements, all of the same type as *ElementTypeRef*. These elements are indexed by a non-empty list of values, described through the type list <*IndexTypeRefList*>. Each index type must have the same scalar type. Typically, the index list will contain one value, describing a single dimensional array. However, index lists with *n* values, describing *n*-dimensional arrays, may also occur.

- **Records** - Each record type has a declared name *TypeRef*. A record groups together a fixed collection of fields. Each field holds a single value referenced through *FieldRef* of some type *FieldTypeRef*.

<*Declaration*> ::= *compositeType*(*TypeRef*, <*Composite*>)
<*Composite*> ::= *array*(<*IndexTypeRefList*>, *ElementTypeRef*) |
record(<*FieldList*>)
<*IndexTypeRefList*> ::= [] | [*IndexTypeRef* | <*IndexTypeRefList*>]
<*FieldList*> ::= [] | [*field*(*FieldRef*, *FieldTypeRef*) | <*FieldList*>]

Figure 6.5: Composite types

Declarations from FDL

`type onecolumn = array [integer] of integer;`
`type onescreen = array [integer] of onecolumn;`
`type teletextpage = record`
`status : pagestatus;`
`content : onescreen`
`end;`
`type teletextpages = array [integer] of teletextpage;`

Proof planner structures
compositeType(*onecolumn*, *array*([*integer*], *integer*)) *compositeType*(*onescreen*, *array*([*integer*], *onecolumn*)) *compositeType*(*teletextpage*, *record*([*field*(*status*, *pagestatus*), *field*(*content*, *onescreen*)])) *compositeType*(*teletextpages*, *array*([*integer*], *teletextpage*))

Figure 6.6: SumOnAirTeletextPages composite types

Constants

A constant is an identifier with a static value. Constants can be created for all types. In particular, the end points of numeric type ranges are expressed as constants. The structure for storing constants is shown in Figure 6.7. Each constant has a declared name *ConstantRef* and is of type *TypeRef*. For illustration, the corresponding structures for the SumOnAirTeletextPages subprogram are shown in Figure 6.8. Note that the constant value itself is not recorded. This echos the SPARK Approach, where constant values are provided indirectly through a subprogram rules (RLS) file. This style is advantageous, as it separates the declaration of entities from the declaration of expressions.

<*Declaration*> ::= *constant*(*ConstantRef*, *TypeRef*)

Figure 6.7: Constants

Variables

A variable is an identifier that can take different values during the execution of a program. Variables can be created for all types. The structure for holding variables is shown in Figure 6.9. Each variable has a declared name *VariableRef* and is of type *TypeRef*. Each variable is associated with a category <*VariableCategory*> as described below:

- **Subprogram variables** - Subprogram variables have category *subprogram*. These correspond to the program variables that are visible within a subprogram.
- **Auxiliary variables** - Auxiliary variables may be introduced, alongside any relevant contextual information, through the category *auxiliary*(<*ContextList*>). Auxiliary variables are introduced to support program analysis.

Declarations from FDL
`const total__last : integer = pending;`
`const total__first : integer = pending;`
`const columns__last : integer = pending;`
`const columns__first : integer = pending;`
`const rows__last : integer = pending;`
`const rows__first : integer = pending;`
`const pagestatus__last : pagestatus = pending;`
`const pagestatus__first : pagestatus = pending;`
`const pagesindex__last : integer = pending;`
`const pagesindex__first : integer = pending;`
`const short_short_integer__last : integer = pending;`
`const short_short_integer__first : integer = pending;`
`const integer__last : integer = pending;`
`const integer__first : integer = pending;`

Proof planner structures	
constant(*firstpage*, *integer*)	*constant*(*total_base_last*, *integer*)
constant(*total_last*, *integer*)	*constant*(*total_base_first*, *integer*)
constant(*total_first*, *integer*)	*constant*(*columns_base_last*, *integer*)
constant(*columns_last*, *integer*)	*constant*(*columns_base_first*, *integer*)
constant(*columns_first*, *integer*)	*constant*(*rows__base__last*, *integer*)
constant(*rows__last*, *integer*)	*constant*(*rows__base__first*, *integer*)
constant(*rows__first*, *integer*)	*constant*(*pagestatus_base_last*, *pagestatus*)
constant(*pagestatus_last*, *pagestatus*)	*constant*(*pagestatus_base_first*, *pagestatus*)
constant(*pagestatus_first*, *pagestatus*)	*constant*(*pagesindex_base_last*, *integer*)
constant(*pagesindex_last*, *integer*)	*constant*(*pagesindex_base_first*, *integer*)
constant(*pagesindex_first*, *integer*)	*constant*(*short_short_integer_base__last*, *integer*)
constant(*short_short_integer_last*, *integer*)	*constant*(*short_short_integer_base__first*, *integer*)
constant(*short_short_integer_first*, *integer*)	*constant*(*integer__base_last*, *integer*)
constant(*integer_last*, *integer*)	*constant*(*integer__base_first*, *integer*)
constant(*integer_first*, *integer*)	

Figure 6.8: SumOnAirTeletextPages constants

For illustration, the corresponding structures[2] for the SumOnAirTeletextPages subprogram are shown in Figure 6.10.

<*Declaration*> ::= *variable*(<*VariableCategory*>, *VariableRef*, *TypeRef*)
<*VariableCategory*> ::= *subprogram* | *auxiliary*(<*ContextList*>)
<*ContextList*> ::= [] | [*Context* | <*ContextList*>]

Figure 6.9: Variables

Declarations from FDL

`var loop__1__i : integer;`
`var r : integer;`
`var tp : teletextpages;`

Proof planner structures
variable(*subprogram*, *i*, *integer*) *variable*(*subprogram*, *r*, *integer*) *variable*(*subprogram*, *tp*, *teletextpages*)

Figure 6.10: SumOnAirTeletextPages variables

Functions

Various mathematical functions may be introduced. The structure for holding functions is shown in Figure 6.11, and the functions it holds are described below.

- **Functions** - Each function has a declared name *FunctionRef*. The function may have zero or more arguments, described through the type list <*ArgTypeRefList*>, and returns a value of type *ReturnTypeRef*. Each function is associated with a category <*FunctionCategory*> as described below:
  - **Built-in functions** - Functions with category *builtin* are implicitly declared. These functions correspond to standard functions, with predefined definitions and properties. All of the built-in functions required to express the VCs considered in the thesis are shown in §6.5.2.
  - **Subprogram functions** - As observed in §4.3.1, SPARK function subprograms are pure mathematical functions. Functions with category *subprogram* are explicitly declared to support reasoning about these functions. The functions may be defined through a user rule (RLU) file.

[2] In the SPARK Approach, the for-loop iterator *i* is referenced as *loop_*<*counter*>*_i*. This name is guaranteed to be unique within its enclosing subprogram. For brevity, in the examples shown in this thesis, every for-loop iterator is uniquely referenced via its program variable name.

– **Auxiliary functions** - Auxiliary functions may be introduced through the category *auxiliary*. Auxiliary functions are introduced to support program analysis, as described in §7.6.4 and §G.8.

<*Declaration*> ::= *function*(<*FunctionCategory*>,
FunctionRef, <*ArgTypeRefList*>, *ReturnTypeRef*)
<*FunctionCategory*> ::= *builtin* | *subprogram* | *auxiliary*
<*ArgTypeRefList*> ::= [] | [*ArgTypeRef* | <*ArgTypeRefList*>]

Figure 6.11: Functions

6.5.2 Built-in Functions

The built-in functions introduce standard operations that are required in reasoning about subprograms. All of the built-in functions relevant to this thesis are described below. Note that each function is associated with its actual syntax, as used by the SPARK Approach, and an alternative syntax for presentation purposes.

Arithmetic Functions

The arithmetic functions support numeric operations, as summarised in Figure 6.12. Addition, subtraction, multiplication, exponentiation and unary minus are available for integer and real types with the usual meanings. Real division may only be evaluated where the result is an exact value. Integer division truncates toward zero. Modulus is available for integer types, and is defined entirely in terms of integer division, as show below:

$$(Y \; mod \; Z) = (Y - ((Y \; div \; Z) * Z)) \tag{6.1}$$

Operation	Actual Syntax	Alternative Syntax
addition	`Y + Z`	$Y + Z$
subtraction	`Y - Z`	$Y - Z$
multiplication	`Y * Z`	$Y * Z$
exponentiation	`Y ** Z`	$Y ** Z$
minus	`- Z`	$-Z$
integer division	`Y div Z`	$Y \; div \; Z$
integer modulus	`Y mod Z`	$Y \; mod \; Z$
real division	`Y / Z`	Y/Z

Figure 6.12: Arithmetic functions

Boolean Functions

The Boolean functions support logical operations, as summarised in Figure 6.13. Conjunction, disjunction, implication, equivalence and negation are available for Boolean types with the usual meanings.

Operation	Actual Syntax	Alternative Syntax
conjunction	`Y and Z`	$Y \wedge Z$
disjunction	`Y or Z`	$Y \vee Z$
implication	`Y -> Z`	$Y \rightarrow Z$
equivalence	`Y <-> Z`	$Y \leftrightarrow Z$
negation	`not Z`	$\neg Z$

Figure 6.13: Boolean functions

Relational Functions

The relational functions support comparison operations, as summarised in Figure 6.14. Equality and its negation are available for all types, including Boolean. The inequality operations less-than, less-than-or-equal, greater-than and greater-than-or-equal, are available for all scalar types, except Boolean, with the usual meanings.

Operation	Actual Syntax	Alternative Syntax
equality	`Y = Z`	$Y = Z$
not equal	`Y <> Z`	$Y \neq Z$
less-than	`Y < Z`	$Y < Z$
less-than-or-equal-to	`Y <= Z`	$Y \leq Z$
greater-than	`Y > Z`	$Y > Z$
greater-than-or-equal-to	`Y >= Z`	$Y \geq Z$

Figure 6.14: Relational Functions

Array Manipulation Functions

The array manipulation functions support the access and updating of array variables, as summarised in Figure 6.15. Array access returns the element of array *ArrayRef* at index *IndexList*. Array update returns array *ArrayRef* with the value of the element at index *IndexList* replaced with *Element*.

Operation	Actual Syntax	Alternative Syntax
array access	`element(ArrayRef, IndexList)`	*element*(*ArrayRef*, *IndexList*)
array update	`update(ArrayRef, IndexList, Element)`	*update*(*ArrayRef*, *IndexList*, *Element*)

Figure 6.15: Array Manipulation Functions

Record Manipulation Functions

The record manipulation functions support the access and updating of record variables, as summarised in Figure 6.16. Record field access returns the value of field *FieldRef* for record *RecordRef*. Record field update returns record *RecordRef* with the value of field *FieldRef* replaced with *Value*. Note that these functions are named after the fields they access, rather than accepting fields as parameters.

Operation	Actual Syntax	Alternative Syntax
record field access	`fld__FieldRef(RecordRef)`	*fld_FieldRef*(*RecordRef*)
record field update	`upf__FieldRef(RecordRef, Value)`	*upf_FieldRef*(*RecordRef*, *Value*)

Figure 6.16: Relational Functions

Quantification Functions

The quantification functions support the description of a collection of properties, as summarised in Figure 6.17. Universal quantification describes a property *Prop* that holds for all values of the quantified variable *QVar* of type *TypeRef*. Existential quantification describes a property *Prop* that holds for at least one value of a quantified variable *QVar* of type *TypeRef*. Note that, in the SPARK Approach, it is very common for the property to take the form of an implication *Guard* $\rightarrow$ *GuardedProp*, where *Guard* constrains the values of the quantified variable *QVar*.

Operation	Actual Syntax	Alternative Syntax
universal quantification	`for_all(QVar:TypeRef, Prop)`	$\forall$(*QVar* : *TypeRef*. *Prop*)
existential quantification	`for_some(QVar:TypeRef, Prop)`	$\exists$(*QVar* : *TypeRef*. *Prop*)

Figure 6.17: Quantification Functions

6.5.3 Goal Structure

The structure for holding goals is shown in Figure 6.18. Each goal has a unique identifier *GoalId*. In the SPARK Approach the conclusions of a VC are grouped together because they occur at the same point in a program. Typically, the conclusions of a VC relate to different proof obligations that are susceptible to different reasoning strategies. For this reason, it is more appropriate to plan each conclusion of a VC as a separate goal. Thus, each proof planning goal contains a single conclusion *Conc* and its corresponding hypotheses <*HypList*>. Meta-logical facts are associated with each goal through the global contextual information <*GlobalContextList*>. The provenance of each goal is recorded to facilitate integration with the SPARK Approach:

- *sourceSubprogram*(*SubprogramName*) - *SubprogramName* is the name of subprogram being verified.
- *sourceFile*(*FileName*) - *FileName* is the name of the file that contains the VC associated with this goal.
- *sourceSystem*(<*FileKind*>) - <*FileKind*> describes whether the VC occurs before simplification, as *vcg*, or after simplification as *siv*.
- *sourceVC*(*VCId*) - *VCId* is the identifier of the VC associated with this goal.
- *sourceConc*(*ConcId*) *ConcId* is the identifier of the conclusion associated with this goal.

In selecting a strategy to prove a goal it is beneficial to know how the goal relates to the source code. For this reason, each goal is associated with its traceability information:

- *traceInfo*(<*Trace*>) - The traceability information associated with the VC corresponding to the goal is described through <*Trace*>. The attributes of this structure are shown in Figure 6.18.

Our techniques target those goals that are not already proved by the SPARK Approach. For this reason, each goal is associated with its current proof status:

- *provedAtSimplifier*(*Boolean*) - Describes whether or not the goal was proved following an application of the Simplifier.

During the planning of a goal, information may be acquired that is applicable to the entire plan. Such information is also attached to the global contextual information:

- *underConstrainedVars*(*VarList*) - During proof planning, it is possible to identify the variables that contribute to a branch of reasoning. If the branch of reasoning ends in failure, its contributing variables may be under constrained. Such potentially under constrained variables are recorded, supporting richer failure analysis.

For illustration, consider the SumOnAirTeletextPages subprogram. Eight VCs must be proved to verify that this subprogram is free from exceptions. One VC corresponds to proving that, at line 9, *i* is a legal index of array *tp*. This VC, and the goal corresponding to its first conclusion, is shown in Figure 6.19. Note that the trivial goal is recorded as having been proved by the Simplifier.

<*Goal*> ::= *goal*(*GoalId*, <*GlobalContextList*>, <*HypList*>, *Conc*)
<*HypList*> ::= [] | [*Hyp* | <*HypList*>]
<*GlobalContextList*> ::= [] | [*GlobalContext* | <*GlobalContextList*>]
<*GlobalContext*> ::= *sourceSubprogram*(*SubprogramName*) |
sourceFile(*FileName*) |
sourceSystem(<*FileKind*>) |
sourceVC(*VCId*) |
sourceConc(*ConcId*) |
traceInfo(<*Trace*>) |
provedAtSimplifier(*Boolean*) |
underConstrainedVars(*VarList*)
<*FileKind*> ::= *vcg* | *siv*
<*Trace*> ::= *betweenPath*(<*FromCut*>, <*ToCut*>) | *refinementIntegrity*
<*FromCut*> ::= *start* | *assertion*(<*CutKind*>, *LineInt*)
<*ToCut*> ::= *finish* | *assertion*(<*CutKind*>, *LineInt*) | *check*(<*CheckKind*>, *LineInt*)
<*CutKind*> ::= *userdefined* | *sparkdefined*
<*CheckKind*> ::= *userdefined* | *runtime* | *precondition*

Figure 6.18: Goals

VC from VCG file (relating to line 9 of Figure 6.2)

```
For path(s) from assertion of line 8 to run-time check associated with
statement of line 9:

procedure_sumonairteletextpages_5.
H1:    r >= 0 .
H2:    r <= loop__1__i - pagesindex__first .
H3:     for_all(i___3: integer, ((i___3 >= columns__first) and (
           i___3 <= columns__last)) -> ( for_all(i___2:
           integer, ((i___2 >= rows__first) and (i___2 <=
           rows__last)) -> ( for_all(i___1: integer, ((
           i___1 >= pagesindex__first) and (i___1 <=
           pagesindex__last)) -> ((element(element(fld_content(element(
           tp, [i___1])), [i___2]), [i___3]) >=
           short_short_integer__first) and
           (element(element(fld_content(element(
           tp, [i___1])), [i___2]), [i___3]) <=
           short_short_integer__last)))))) .
H4:     for_all(i___1: integer, ((i___1 >= pagesindex__first) and (
           i___1 <= pagesindex__last)) -> ((fld_status(element(
           tp, [i___1])) >= pagestatus__first) and (fld_status(element(
           tp, [i___1])) <= pagestatus__last))) .
H5:    loop__1__i >= pagesindex__first .
H6:    loop__1__i <= pagesindex__last .
H7:    loop__1__i <= pagesindex__last .
        ->
C1:    loop__1__i >= pagesindex__first .
C2:    loop__1__i <= pagesindex__last .
```

Proof planner structures (corresponding to first conclusion of above VC)

$$
\begin{array}{l}
goal(goal(24),\\
\quad [sourceSubprogram(sumonairteletextpages), sourceFile(sumonairteletextpages.vcg),\\
\quad sourceSystem(vcg), sourceVC(5),\\
\quad traceInfo(betweenPath(assertion(userdefined, 8), check(runtime, 9))),\\
\quad sourceConc(1), provedAtSimplifier(true)],\\
\quad [r \geq 0, r \leq (i - firstpage),\\
\quad \forall(i\_3 : integer.\ i\_3 \geq columns\_first \wedge i\_3 \leq columns\_last \rightarrow\\
\qquad \forall(i\_2 : integer.\ i\_2 \geq rows\_first \wedge i\_2 \leq rows\_last \rightarrow\\
\qquad\quad \forall(i\_1 : integer.\ i\_1 \geq pagesindex\_first \wedge\\
\qquad\qquad i\_1 \leq pagesindex\_last \rightarrow\\
\qquad element(element(fld\_content(element(tp, [i\_1])), [i\_2]), [i\_3]) \geq short\_short\_integer\_first \wedge\\
\qquad element(element(fld\_content(element(tp, [i\_1])), [i\_2]), [i\_3]) \leq short\_short\_integer\_last))),\\
\quad \forall(i\_1 : integer.\ i\_1 \geq pagesindex\_first \wedge i\_1 \leq pagesindex\_last \rightarrow\\
\qquad fld\_status(element(tp, [i\_1])) \geq pagestatus\_first \wedge\\
\qquad fld\_status(element(tp, [i\_1])) \leq pagestatus\_last),\\
\quad i \geq pagesindex\_first,\\
\quad i \leq pagesindex\_last,\\
\quad i \leq pagesindex\_last],\\
\quad i \geq pagesindex\_first)
\end{array}
$$

Figure 6.19: SumOnAirTeletextPages selected VC and goal

6.6 Theorems

Properties and definitions are held in external rule files. These files are retrieved as theorems, as described in §6.6.1. Subsequently, these theorems are processed to generate a collection of rewrite rules for use during proof planning, as described in §6.6.2.

6.6.1 Retrieving Theorems

Three different categories of rules may be available for each subprogram, as considered below:

- **Subprogram rules (RLS)** - The subprogram rules are specific to the entities and types in the subprogram. Every subprogram rule is transformed into a theorem.
- **Standard rules (RUL)** - The standard rules are available to all subprograms. Some standard rules cannot be expressed as strictly logical theorems as they contain computational guards. Further, some standard rules are obviously not relevant where verifying exception freedom. Thus, the strictly logical rules from a core subset of standard rule files are transformed into theorems. The selected subset comprises the rule files *arith*, *fdlfuncs*, *genineqs*, *intineqs*, *logic*, *modular* and *numineqs* as detailed in [Praa].
- **User rules (RLU)** - The user rules are available to all subprograms. These rules are created by an engineer to provide additional definitions and proprieties. Every user rule is transformed into a theorem. Some user rules are introduced to support the verification of exception freedom, as discussed in §B.4.

The structure for holding theorems is shown in Figure 6.20. Each theorem has a unique identifier *TheoremId* and is presented as the expression *TheoremExp*. The theorems are extracted from rule files in accordance with [Praa], as summarised in Figure 6.21. The provenance of each theorem is recorded to facilitate integration with the SPARK Approach. Each theorem is held alongside its source file *FileName*, its file kind <*FileKind*> and its rule identifier *RuleId*.

<*Theorem*> ::= *theorem*(*TheoremId*, *TheoremExp*,
FileName, <*FileKind*>, *RuleId*)
<*FileKind*> ::= | *rul* | *rlu* | *rls*

Figure 6.20: Proof planner theorems

Note that rule files may be preceded by a type header. These headers are difficult to work with and, in general, are not sufficiently strong to convey the type of all rules. Thus, the type headers are completely ignored. Instead, a few simple heuristics are used to infer the type of rules. In principle, this weakness might allow the proof planner to discover a

proof plan which is ultimately rejected by the Checker. In practice, this situation has not occurred as the type inference heuristics are generally effective.

Rule	**Theorem expression**
X `may_be_replaced_by` Y	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (true \rightarrow (x = y)))$
X `may_be_replaced_by` Y `if` [*CList*]	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (c \rightarrow (x = y)))$
X & Y `are_interchangeable`	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (true \rightarrow (x = y)))$
X & Y `are_interchangeable if` [*CList*]	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (c \rightarrow (x = y)))$
X `may_be_deduced`	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (true \rightarrow (true \rightarrow x)))$
X `may_be_deduced if` [*CList*]	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (c \rightarrow (true \rightarrow x)))$
X `may_be_deduced_from` [*YList*]	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (true \rightarrow (y \rightarrow x)))$
X `may_be_deduced_from` [*YList*] `if` [*CList*]	$\forall(v_1 : t_1 \ldots v_i : t_i.\ (c \rightarrow (y \rightarrow x)))$

Lists of conjuncts *YList* and *CList* are conjoined to form the expressions Y and C respectively. Every implicitly quantified variable in X, Y and C is replaced with explicitly quantified variables $v_1 \ldots v_i$ of appropriate type $t_1 \ldots t_i$ to give x, y and c. Note that *true* is inserted such that every theorem takes either the form $(U \rightarrow (V = W))$ or $(U \rightarrow (V \rightarrow W))$.

Figure 6.21: Converting from rules to theorems

6.6.2 Converting Theorems to Rewrite Rules

Each theorem is transformed into a pair of rewrite rules. The structure for holding rewrite rules is shown in Figure 6.22. Each rewrite rule has a unique identifier *RewriteRuleId* and is associated with its corresponding theorem *SourceTheoremId*. Key details of the rewrite rule are provided through its direction and polarity as described below:

- <*Direction*> - Records how the theorem was oriented in generating the rewrite rule. Where the rewrite rule is oriented in the same direction as the theorem, the direction is *normal* and *reversed* otherwise. Note that this information is required in extracting a compound tactic from an instantiated proof plan.

- <*Polarity*> - Describes the logical contexts in which the rewrite rule may be applied. A conclusion is considered to be of *positive* polarity and a hypothesis of *negative* polarity. A rule marked as *zero* polarity may be applied to any subexpression, a rule marked as *positive* must be applied to a *positive* subexpression and a rule marked as *negative* must be applied to a *negative* subexpression. Note that, in our proof plans, all polarity concerns are centrally handled by the predicate sub_exp_polarity as described in §C.9.17.

The rewrite rule itself is presented with a conditional guard as *Condition* and rewriting from expression *LHSExp* to expression *RHSExp*. Each theorem takes a standard form, as noted in Figure 6.21, thus the transformation from theorems to rewrite rules is straight forward, as described in Figure 6.23. For illustration, Figure 6.24 shows a subset of the

rules generated for the SumOnAirTeletextPages subprogram alongside their corresponding theorem and rewrite rule structures.

<*RewriteRule*> ::= *rewriteRule*(*RewriteRuleId*, *SourceTheoremId*,
<*Direction*>, <*Polarity*>,
Condition : *LHSExp* ⇒ *RHSExp*)
<*Direction*> ::= *normal* | *reversed*
<*Polarity*> ::= *zero* | *positive* | *negative*

Figure 6.22: Proof planner rewrite rules

The universally quantified variables in each theorem are unwrapped and replaced with meta-variables. The resulting expressions take the form of an equality or implication between a left hand side expression *LHSExp* and a right hand side expression *RHSExp* guarded by a condition *Condition*, as follows:

$$Condition \rightarrow (LHSExp = RHSExp) \tag{6.2}$$

$$Condition \rightarrow (LHSExp \rightarrow RHSExp) \tag{6.3}$$

The mapping from these expressions to their corresponding pair of rewrite rules and associated direction and polarity is shown in the tables below:

Unwrapped theorem expression
Condition → (*LHSExp* = *RHSExp*)

⇓

Rewrite rule	*Direction*	*Polarity*
Condition : *LHSExp* ⇒ *RHSExp*	*normal*	*zero*
Condition : *RHSExp* ⇒ *LHSExp*	*reversed*	*zero*

Unwrapped theorem expression
Condition → (*LHSExp* → *RHSExp*)

⇓

Rewrite rule	*Direction*	*Polarity*
Condition : *LHSExp* ⇒ *RHSExp*	*normal*	*negative*
Condition : *RHSExp* ⇒ *LHSExp*	*reversed*	*positive*

Figure 6.23: Converting from theorems to rewrite rules

Subset of RLS file

```
sumonairtele_rules(3): integer__first may_be_replaced_by -2147483648.
sumonairtele_rules(4): integer__last may_be_replaced_by 2147483647.
sumonairtele_rules(16): pagestatus__last may_be_replaced_by offair.
sumonairtele_rules(26): pagestatus__pos(pred(X)) may_be_replaced_by pagestatus__pos(X) - 1 if
    [X >=onair, X <> onair].
```

Corresponding subset of theorem structures

$theorem(theorem(3), firstpage = 100,$
$\quad sumonairteletextpages.rls, rls, sumonairtele\_rules(3))$
$theorem(theorem(4), integer\_first = -2147483648,$
$\quad sumonairteletextpages.rls, rls, sumonairtele\_rules(4))$
$theorem(theorem(16), pagestatus\_first = onair,$
$\quad sumonairteletextpages.rls, rls, sumonairtele\_rules(16))$
$theorem(theorem(26), \forall(x : pagestatus.\ x \leq offair \wedge x \neq offair \rightarrow$
$\quad pagestatus\_pos(succ(x)) = pagestatus\_pos(x) + 1),$
$\quad sumonairteletextpages.rls, rls, sumonairtele\_rules(26))$

Corresponding subset of rewrite rule structures

$rewriteRule(rr(5), theorem(3), normal, zero, true : firstpage \Rightarrow 100)$
$rewriteRule(rr(6), theorem(3), reversed, zero, true : 100 \Rightarrow firstpage)$
$rewriteRule(rr(7), theorem(4), normal, zero, true : integer\_first \Rightarrow -2147483648)$
$rewriteRule(rr(8), theorem(4), reversed, zero, true : -2147483648 \Rightarrow integer\_first)$
$rewriteRule(rr(31), theorem(16), normal, zero, true : pagestatus\_first \Rightarrow onair)$
$rewriteRule(rr(32), theorem(16), reversed, zero, true : onair => pagestatus\_first)$
$rewriteRule(rr(51), theorem(26), normal, zero, X \leq offair \wedge X \neq offair :$
$\quad pagestatus\_pos(succ(X)) \Rightarrow pagestatus\_pos(X) + 1)$
$rewriteRule(rr(52), theorem(26), reversed, zero, X \leq offair \wedge X \neq offair :$
$\quad pagestatus\_pos(X) + 1 \Rightarrow pagestatus\_pos(succ(X)))$

Figure 6.24: Subset of SumOnAirTeletextPages rules

6.7 Proof Plans

A proof plan is expressed through *methods* and *critics*, as described in §6.7.1. In our proof planner, method applications are controlled through *strategies*, as described in §6.7.2.

6.7.1 Methods and Critics

Each method is stored in a *method* (MTD) file. The general form of a method is show in Figure 6.25. The purpose of each method slot is described below:

- **Method** - The name of the method.
- **Tactic** - Describes how to perform the actions of the method at the object-level. The tactic typically contains meta-variables which will become instantiated during a successful method invocation.
- **Goal** - The goal inputted to the method. The slot may act as an additional method precondition by only accepting goals that match a specific pattern.
- **Preconditions** - Preconditions constrain the application of the method. Where the preconditions are successful the method is expected to be successful. The preconditions are expressed in the meta-level theory through a method-language.
- **Effects** - Effects perform the actions of the method. The effects are only performed if the method preconditions hold. The effects are expressed in the meta-level theory through a method-language.
- **Subgoals** - The potentially multiple subgoals outputted by the method. A terminating method will generate an empty list of subgoals.

Method:
Method
Tactic:
Tactic
Goal:
$LocalContextList : HypList \vdash Conc$
Preconditions:
$[Precondition_1, \ldots, Precondition_x]$
Effects:
$[Effect_1, \ldots, Effect_y]$
Subgoals:
$[Subgoal_1, \ldots, Subgoal_z]$

Figure 6.25: Method template

Each method may have a number of corresponding critics. Each critic is stored in a *critic* (CRI) file. The general form of a critic is show in Figure 6.26. The purpose of each critic slot is described below:

- **Critic** - The name of the critic.
- **Parent method** - The name of the method that the critic is associated with.
- **Goal** - The goal inputted to the critic. The slot may act as an additional critic precondition by only accepting goals that match a specific pattern.
- **Successful method preconditions** - The method preconditions that must have been successful for the critic to be triggered.
- **Failed method precondition** - The method precondition that must have failed, or not been explored, for the critic to be triggered.
- **Preconditions** - Preconditions constrain the application of the critic. Where the preconditions are successful the critic is expected to be successful. The preconditions are expressed in the meta-level theory through a method-language.
- **Effects** - Effects perform the actions of the critic. The effects are only performed if the critic preconditions hold. The effects are expressed in the meta-level theory through a method-language.

Critic:
Critic
Parent Method:
ParentMethod
Goal:
$LocalContextList : HypList \vdash Conc$
Successful method preconditions:
$[SuccessfulPrecondition_1, \ldots, SuccessfulPrecondition_x]$
Failed method precondition:
FailedPrecondition
Preconditions:
$[Precondition_1, \ldots, Precondition_y]$
Effects:
$[Effect_1, \ldots, Effect_z]$

Figure 6.26: Critic template

6.7.2 Strategies

A proof plan is constructed from a number of methods. Thus, there is a need for a mechanism to control method applications. Various different mechanisms have been considered

[DJP06]. Modern proof planners, such as IsaPlanner [Dix05], use powerful *methodicals*, extending the method-language to include method-level operations. For simplicity, our proof planner controls method applications through a weaker mechanism called *strategies*. These strategies are held in a *strategy* (STR) file.

The general form of a strategy is show in Figure 6.27. Each strategy contains a set of *waterfalls*. A waterfall is a named ordered list of *actions*. Each action references a method to try, and, where that method is successful, the waterfall to invoke on its subgoals. For generality, the actions associated with a waterfall may be parametrised[3].

Waterfall:
$WaterfallRef_1(Parameters)$
Actions:
$MethodRef_1 \mapsto SelWaterfallRef(SelParameters)$
...
$MethodRef_y \mapsto SelWaterfallRef(SelParameters)$

...

Waterfall:
$WaterfallRef_x(Parameters)$
Actions:
$MethodRef_1 \mapsto SelWaterfallRef(SelParameters)$
...
$MethodRef_z \mapsto SelWaterfallRef(SelParameters)$

Figure 6.27: Strategy template

6.8 Proof Planner

The proof planner begins by retrieving the goal, theorems and proof plans. The planner operates on *plans*, as described in §6.8.1. The planner conducts an iterative deepening search, as described in §6.8.2.

6.8.1 Plans

The structure for holding plans is shown in §6.28. A number of plans may be present, each having a unique identifier *PlanId*. Each closed plan is associated with its result *PlanResult*. Each open plan has a proof tree and a search control structure. The proof tree contains the global context information *GlobalContextList* and a collection of nodes of the forms described below:

[3]The parametrisation of waterfalls supports the presentation of common strategies. In practice, our proof planner does not support this parametrisation. The same meaning is achieved by creating specific instantiations of each strategy.

- *goalNode*(*GoalNodeId*, *LocalContextList*, *HypList*, *Conc*) -
  Describes a goal. Each goal node has a unique identifier *GoalNodeId*. The goal is described through its local context information *LocalContextList* and its hypotheses and conclusion as *HypList* and *Conc* respectively.

- *methodNode*(*MethodNodeId*, *MethodName*, *Tactic*,
  ParentGoalNodeId, *ChildrenGoalNodeIdList*) -
  Describes the successful application of a method on a goal. The method nodes join together goal nodes to form the proof tree structure. Each method node has a unique identifier *MethodNodeId*. The corresponding method name and tactic application are recorded as *MethodName* and *Tactic* respectively. The method node connects goal nodes by recording the parent goal that the method was applied to as *ParentGoalId* and any children subgoals that the method generated as *ChildrenGoalIdList*.

Each plan has its own search control, describing every open goal and the current search band as described below:

- *openGoal*(*GoalNodeId*, *DepthInt*, *WaterfallActionList*) -
  Indicates that the goal with identifier *GoalNodeId*, known to be at depth *DepthInt*, has pending method invocations as described by *WaterfallActionList*.

- *searchBand*(*FromInt*, *ToInt*) -
  Indicates that iterative deepening is currently exploring those nodes between depth *FromInt* and depth *ToInt*.

<*Plans*> ::= <*PlanList*>
<*PlanList*> ::= [] | [<*Plan*> | <*PlanList*>]
<*Plan*> ::= *plan*(*PlanId*, <*PlanStatus*>)
<*PlanStatus*> ::= *openPlan*(<*ProofTree*>, <*SearchControl*>) |
closedPlan(*PlanResult*)
<*ProofTree*> ::= *proofTree*(*GlobalContextList*, <*NodeList*>)
<*NodeList*> ::= [] | [*Node* | <*NodeList*>]
<*Node*> ::= *goalNode*(*GoalNodeId*, *LocalContextList*, *HypList*, *Conc*) |
methodNode(*MethodNodeId*, *MethodName*, *Tactic*,
ParentGoalNodeId, *ChildrenGoalNodeIdList*)
<*SearchControl*> ::= *searchControl*(<*OpenGoalList*>, <*SearchBand*>)
<*OpenGoalList*> ::= [] | [*OpenGoal* | <*OpenGoalList*>]
<*OpenGoal*> ::= *openGoal*(*GoalNodeId*, *DepthInt*, *WaterfallActionList*)
<*SearchBand*> ::= *searchBand*(*FromInt*, *ToInt*)

Figure 6.28: Plans

6.8.2 Planner Algorithm

The planner algorithm explores method applications through an iterative deepening search [Kor85], as described below. Note that this algorithm is similar to the iterative deepening search available in Clam [DRe06].

- **Initialisation** - At initialisation, a *main* plan is created for the input goal. The goal is added as the *root* goal node in the proof tree. Any global context information associated with the goal is added to the proof tree. The search control is initialised, indicating that the root goal is open. Unless stated otherwise, the initial strategy is exception_freedom, as described in §E.3.

- **Planning** - While open plans remain, the selection and application phases described below are repeatedly performed.

  - **Selection** - A plan, goal and method is selected.

    * **Select Plan** - The first open plan is selected.
    * **Select Goal** - The deepest open goal within the current band of the iterative deepening search is selected. Where there are multiple goals at this depth, the goal that has been unexplored for longest is selected next. If there are no open goals in the current band then the band is increased to explore the next 3 depths. If the search band exceeds depth 70, then the plan is closed with the failure critique *maximumDepthReached*. The search band was kept small, as successful plans often occur at a relatively shallow depth. The depth limit was established empirically, comfortably holding all of the plans we have encountered. If no open goals can be found then the plan is closed with a suitable failure critique. If under constrained variables are associated with the plan then the failure critique will report these via an *abstractPredicate* suggesting *tightlyConstrainVars*. Otherwise, the failure critique will report *noMoreOpenGoals*.
    * **Select Method** - The waterfall list associated with the open goal is queried. The next action is selected and removed from the list. If this is the last action in the list, then the goal is removed from the list of open goals.

  - **Application** - The selected method is applied to the selected goal of the selected plan.

    * **Method Successful** - The proof tree is extended to record the method application and any subgoals that were generated. Where the method generates no subgoals it is checked to see if a proof for the root goal has been found. If this is the case then the plan is closed, with the result as the discovered instantiated proof plan. Regardless of any pending waterfall actions, the selected goal is always removed from the list of open goals.

Typically, our methods generate subgoals which are closer to completing a proof. Thus, once a method has been successfully applied to a goal there is generally little merit in exploring alternative methods at the same goal. If our proof planner supported methodicals, rather than simple strategies, this efficiency measure could be expressed in the proof plans rather than being embedded in the planner algorithm.

  - **Method Failed** - Where the method fails, any critics associated with the method are attempted. If the plan is aborted by the critic, then the relevant failure critique is reported. If the critic is successful, the planner continues just as if its parent method had been successful. If the critic is unsuccessful, then the method is dismissed.

- **Result** - The result associated with the main plan is returned. The result will either be an instantiated proof plan or a failure critique.

6.9 Plan Result

The proof planner will generate either an instantiated proof plan or a failure critique. From each result the overall result of the proof planner is determined.

6.9.1 Result from Instantiated Proof Plan

Where an instantiated proof plan is discovered its correctness is demonstrated via the Checker. A compound tactic is extracted from the instantiated proof plan. The compound tactic automatically controls the Checker, attempting to prove the relevant goal. Where the goal is proved inside the Checker, a successful proof is reported. Otherwise, failure is reported, highlighting that a defect in the proof plans has been detected. Note that, while the latter case is undesirable, soundness is preserved as the flawed proof is rejected by the Checker.

Proof planners are typically coupled to tactic based theorem provers, as tactics provide a powerful mechanism for describing and executing a discovered proof plan. However, the Checker is not a tactic based theorem prover. To address this mismatch, *simulated* tactics and tacticals are introduced, as shown in Figure 6.29. Tactics perform a unit of reasoning while tacticals supports the composition of tactics. The technique hides the detail associated with interfacing to the Checker, allowing proof plans to be expressed at a natural level of abstraction. The translation from simulated tactics and tacticals into Checker proof commands is detailed in Appendix D.

For illustration, consider the SumOnAirTeletextPages subprogram. Two VCs are generated from the invariant at line 8 back to the same invariant, covering both paths through the if-statement. The VC associated with not entering the if-statement is not proved by the Simplifier. The proof planner discovers an instantiated proof plan for this VC, via

<*CompoundTactic*> ::= <*Tactical*>
<*Tactical*> ::= then_tactical(<*Tactic*>, <*TacticalList*>) |
final_tactical(<*Tactic*>)
<*TacticalList*> ::= [] | [<*Tactical*> | <*TacticalList*>]
<*Tactic*> ::= null_tactic |
trivial_tactic |
trivially_true_conc_tactic(*Conc*) |
rewrite_tactic(*RewriteForm*,
HypOrConc, *WholeExp*, *Pos*,
Condition : *LHSExp* ⇒ *RHSExp*) |
split_conc_conj_tactic(*LeftExp*, *RightExp*) |
case_split_tactic(*FirstExp*, *SecondExp*) |
sequence_tactic(<*TacticList*>)
<*TacticList*> ::= [] | [<*Tactic*> | <*TacticList*>]

Figure 6.29: Simulated tactics and tacticals

the proof plans detailed in Appendix E. The essential details of the VC, its compound tactic and proof commands are show in Figure 6.30. As the compound tactic and proof commands associated with instantiated proof plans are verbose and mechanically derived, they are omitted in all subsequent examples.

6.9.2 Result from Failure Critique

Where a failure critique is raised it is directly reported as the planner result. The structure for every failure critique is shown in Figure 6.31 and described below.

- *maximumDepthReached* - Raised when the proof planner tries to search beyond a fixed depth. Care is taken to express proof plans in a form that is expected to terminate. However, especially during development, a pathological goal may lead to an infinite search.

- *noMoreOpenGoals* - Raised when the proof planner has fully explored a proof plan on a given goal, without making any insights. The proof plan needs to be extended to prove such goals.

- *provedBySimplifier* - Raised where the goal has already been proved by the Simplifier. For further details, see §E.7.

- *simplifiedGoal* - Raised where the goal has been simplified by the Simplifier. For further details, see §E.8.

- *goalNotTargeted* - Raised where goal is of a category that is not targeted by the proof plans. The proof plans will need to be extended to reason about these goals. For further details, see §E.9.

Key portion of invariant goal

$r \leq (i - \mathit{firstpage})$
$\rightarrow$
$r \leq ((i + 1) - \mathit{firstpage})$

Key portion of compound tactic

```
...
then_tactical
rewrite_tactic(rule(ruleref(sumonairteletextpages.rls, rls, sumonairtele_rules(3), normal)),
        hyp, r ≤ (i − firstpage), [2, 2], true : firstpage ⇒ 100)
...
then_tactical
rewrite_tactic(rule(ruleref(sumonairteletextpages.rls, rls, sumonairtele_rules(3), normal)),
        conc, r ≤ (i + 1) − firstpager, [2, 2], true : firstpage ⇒ 100)
...
then_tactical
rewrite_tactic(rule(ruleref(.../user-rules/minusplus.rul, rlu, minusplus(1), reversed)),
        conc, r ≤ (i + 1) − 100, [2], true : (i + 1) − 100 ⇒ (i − 100) + 1
then_tactical
rewrite_tactic(rule(ruleref(.../user-rules/minusplus.rul, rlu, miscellaneous(7), reversed)),
        conc, r ≤ (i − 100) + 1, [], true : r ≤ (i − 100) + 1 ⇒ (r ≤ (i − 100)) ∧ (0 ≤ 1))
then_tactical
rewrite_tactic(hypothesisFertilise(r ≤ (i − 100))
        conc, r ≤ (i − 100) ∧ (0 ≤ 1), [1], true : r ≤ (i − 100) ⇒ true)
then_tactical
rewrite_tactic(evaluate(true ∧ (0 ≤ 1), true)
        conc, true ∧ (0 ≤ 1), [], true : true ∧ (0 ≤ 1) ⇒ true)
final_tactical
trivial_tactic
```

Key portion of proof commands

```
consult 'sumonairteletextpages.rls'.
consult '.../user-rules/minusplus.rul'.
consult '.../user-rules/miscellaneous.rul'.
tame_subgoal_on_conc (2).
    ...
    tame_rewrite (hyp) : (r<=(loop__1__i-pagesindex__first)) : ([2,2]) with
        (pagesindex__first) to (100) if (true) using (sumonairtele_rules(3)) in (normal).
    tame_rewrite (conc) : (r<=(loop__1__i+1)-pagesindex__first) : ([2,2]) with
        (pagesindex__first) to (100) if (true) using (sumonairtele_rules(3)) in (normal).
    ...
    tame_rewrite (conc) : (r<=(loop__1__i+1)-100) : ([2]) with
        ((loop__1__i+1)-100) to ((loop__1__i-100)+1) if (true) using (minusplus(1)) in (reversed).
    tame_rewrite (conc) : (r<=(loop__1__i-100)+1) : ([]) with
        (r<=loop__1__i-100+1) to (r<=loop__1__i-100 and 0<=1) if
            (true) using (miscellaneous(7)) in (reversed).
    tame_rewrite (conc) : (r<=(loop__1__i-100) and 0<=1) : ([1]) where (r<=(loop__1__i-100)).
    tame_rewrite (conc) : (true and 0<=1) : ([]) with (true and 0<=1) is (true).
    tame_done.
tame_all_done.
tame_done.
tame_finish.
```

Figure 6.30: SumOnAirTeletextPages invariant VC (line 8 of Figure 6.2)

- *inRealDomain* - Raised when the conclusion contains fixed or floating point types. The proof planner and proof plans will need to be extended to reason about these types. For further details, see §E.10.

- *abstractPredicate*(*SubprogramName*, *coupleWithEntryVars*(<*VarList*>)) - Raised where a proof plan suggests that properties should be introduced to relate the variables in <*VarList*> with their corresponding loop entry variables in subprogram *SubprogramName.* For further details, see §E.14.

- *abstractPredicate*(*SubprogramName*, *constrainVars*(<*VarList*>)) - Raised where a proof plan suggests introducing constraints on the variables in <*VarList*> in subprogram *SubprogramName.* For further details, see §E.16.

- *abstractPredicate*(*SubprogramName*, *tightlyConstrainVars*(<*VarList*>)) - Raised where a proof plan suggests increasing the constraints on the variables in <*VarList*> in subprogram *SubprogramName.* For further details, see §E.17.

- *interactionNeeded*(*SubprogramName*, *constrainConsts*(<*ConstList*>)) - Raised where a proof plan suggests introducing constraints on the constants in <*ConstList*> in subprogram *SubprogramName.* For further details, see §E.15.

<*FailureCritique*> ::= *maximumDepthReached* |
noMoreOpenGoals |
provedBySimplifier |
simplifiedGoal |
goalNotTargeted |
inRealDomain |
abstractPredicate(*SubprogramName*, <*ProgAnalysisRequest*>) |
interactionNeeded(*SubprogramName*, <*EngineerRequest*>)
<*ProgAnalysisRequest*> ::= *coupleWithEntryVars*(<*VarList*>) |
constrainVars(<*VarList*>) |
tightlyConstrainVars(<*VarList*>)
<*VarList*> ::= [] | [<*Var*> | <*VarList*>]
<*EngineerRequest*> ::= *constrainConsts*(<*ConstList*>)
<*ConstList*> ::= [] | [<*Const*> | <*ConstList*>]

Figure 6.31: Failure Critique structure

6.10 Method-Language Overview

A method-language is developed to support the verification of exception freedom in the SPARK Approach. The method-language predicates are grouped into a number of categories, as summarised below. Full details of the method-language are presented in Appendix C.

- **Composition** (§C.3) - These predicates support the composition of methods. The predicates are typically available in proof planners.
- **Proof Planning** (§C.4) - These predicates offer direct control of the planning process. As described in §6.7.2, method applications are controlled through a relatively limited strategy mechanism. By allowing methods to directly control the planning process, some of these limitations are addressed.
- **List Processing** (§C.5) - These predicates support list processing. Predicates tend to operate with lists rather than individual elements to increase generality and support reuse. Thus, there is a general need for list processing predicates.
- **Plan Features** (§C.6) - These predicates allow methods to query and modify the global contextual information associated with each plan.
- **Goal Features** (§C.7) - These predicates allow methods to query and modify the local contextual information associated with each goal.
- **Goal Patterns** (§C.8) - In developing methods for verifying exception freedom, a number of relevant goal patterns emerged. These predicates support the detection and evaluation of these patterns.
- **Analyse Expressions** (§C.9) - These predicates support the manipulation of expressions. Such methods are typically available in proof planners.
- **Rewriting** (§C.10) - These predicates support term rewriting, a powerful theorem proving technique. General predicates support rewriting operations. Further, a few specialised predicates provide efficient access to operations that are achieved through multiple rewrites.
- **Rippling** (§C.11) - These predicates support an application of the rippling heuristic. As described in §2.3.3, the rippling heuristic is directly applicable to the proof of loop invariant goals.

6.11 Proof Plans Overview

Proof plans are developed to support the verification of exception freedom in the SPARK Approach. An overview of the plans developed for exception freedom goals and program analysis queries are given in §6.11.1 and §6.11.2 respectively. Full details of the plans are presented in Appendix E.

6.11.1 Proof Plans for Exception Freedom Goals

Three proof plan strategies support the verification of exception freedom goals, as summarised below.

- exception_freedom (§E.3) - Given the context of the SPARK Approach, a number of goals may not be relevant to advancing the verification of exception freedom. Thus, the targeted_goal method (§E.6) filters out irrelevant goals. The Examiner tends to generate verbose goals. Thus an initialisation method (§E.11) is applied early, transforming the goal into a more readily analysable form. In proving exception freedom, goals often have general hypotheses and specific conclusions. In many cases, obvious specialisations of the general hypotheses will contribute toward proof. The specialise_hyps method (§E.12) performs these obvious specialisations. During proof planning, it is difficult to determine if a false goal has arisen because the original goal is not provable or because a poor choice of proof step was made. Thus, prior to any proof exploration, the viable_goal method (§E.13) investigates the provability of the goal. The method searches for patterns that are commonly associated with unprovable goals. This includes the use of a constraint solver to identify counter examples that demonstrate the goal is not provable. Different goal categories may be identified depending on how the goal relates to the source code. Significantly, different goal categories are susceptible to different proof strategies. Two different goal categories are considered. The *run-time check* category is identified by the rtc_goal method (§E.18) while the *invariant* category is identified by the inv_goal method (§E.19).

- run_time_check (§E.4) - All goals corresponding to the run-time check category will involve proving that, given various constraints, a particular bound is never violated. These goals are addressed through a collection of cooperating techniques. The goal is simplified to identify its essential details, as achieved by the true_conc method (§E.20), the false_conc method (§E.21), the eval_conc method (§E.25), the split_conc_conj method (§E.26), the fertilize method (§E.27), the clear_conc_exp method (§E.28) and the elim_var_conc method (§E.29). The automated capabilities of the Checker are exploited to perform the simplification of linear functions, as performed by the linear_bounded_conc method (§E.22). To more readily exploit the standard rules supplied with the SPARK Approach, multiplication is normalised by the case_split method (§E.23) and the mult_commute method (§E.24). Where the above techniques are unable to prove a goal, the conclusion is decomposed into more tractable expressions. The decomposition is achieved through an application of transitivity, as initiated by the transitivity_entry method (§E.30). The process requires a creative eureka step, discovering an intermediate expression that supports the application of a transitivity rule. The search is achieved through middle-out reasoning, as supported by the transitivity_fertilize method (§E.32), the transitivity_decomp method (§E.31), the transitivity_close method (§E.33) and the transitivity_unblock method (§E.34).

- invariant (§E.5) - All goals corresponding to the invariant category involve proving that an invariant property is preserved following an iteration of a loop. As

these goals exhibit the same pattern as proof by induction the rippling heuristic is directly applicable. Thus, the rippling heuristic is reused, as achieved by the ripple_entry method (§E.35), the ripple_unblock method (§E.38), the ripple_wave method (§E.36) and the ripple_fertilize method (§E.37). Following a successful application of rippling it is common for a proof residue to remain. As the invariant properties typically describe invariant bounds, the proof residue often involves proving that a particular bound is not violated. Thus, the strategy developed for run-time check goals is applicable and is directly reused.

6.11.2 Proof Plans for Program Analysis Queries

Four proof plan strategies are exploited during program analysis to perform different reasoning queries, as summarised below.

- pa_exp_simplify (§E.40) - Supports the simplification of expressions. Expressions are eliminated through logical properties, as achieved by the prune_conc_duplicate method (§E.44) and the prune_conc_eq method (§E.45). Two simplification techniques developed for run-time check goals are reused, as the eval_conc method (§E.25) and the clear_conc_exp method (§E.28). Where no further simplifications are available, the simplified expression is returned by the report_conc method (§E.46).

- pa_exp_constrain (§E.41) - Supports the generalisation of a complex expression into a weaker, yet simpler, bounded expression. Hypothesis specialisations can contribute toward finding constraints, thus the specialise_hyps method (§E.12) is reused. The resulting constraints are enriched through the services of a computer algebra system, via the solve_eq_hyp_for_var method (§E.47). Finally, with a rich collection of constraints available, the tightest bounded constraint for the complex expression is returned through the constrain_conc_conj method (§E.48).

- pa_spark_exp (§E.42) - Transforms an expression into a form which can be directly expressed in SPARK annotations. The specialise_hyps method (§E.12) is reused to introduce additional hypotheses. Simplifications are performed by reusing the prune_conc_duplicate method (§E.44), the prune_conc_eq method (§E.45), the eval_conc method (§E.25), and the clear_conc_exp method (§E.28). Where applicable, constraints are enriched by reusing the solve_eq_hyp_for_var (§E.47) method. The elim_aux_var_via_eq method (§E.49), the elim_prog_var_exp_via_eq method (§E.50) and the elim_aux_var_via_int_arith method (§E.51) exploit the simplified and enriched constraints, seeking to eliminate expressions that are not expressible in SPARK annotations. Where successful, the resulting SPARK expression is returned through the is_spark_exp method (§E.52).

- pa_disj_norm_form (§E.43) - Converts an expression into disjunctive normal form. The disj_norm_form method (§E.53) applies a rewrite rule which brings an expression closer to disjunctive normal form. Where this method no longer applies, the expression is in disjunctive normal form, and is returned by the report_conc method (§E.46).

Chapter 7

Program Analyser

7.1 Introduction

A program analyser is developed. The program analyser is tailored to the specific task of discovering candidate invariant properties for SPARK subprograms. This chapter describes the details of our program analyser.

7.2 Program Analyser Architecture

The architecture of our program analyser is shown in Figure 7.1. The program analyser is provided with three forms of input. The *MiniSPARK* is the program to be analysed, as described in §7.4. The *abstract predicates* are generated by our proof planner to request specification strengthening, as described in §6.9.2. The *program analysis heuristics* perform the program analysis, as described in §7.9. A *parser* transforms the MiniSPARK into *package information*, as described in §7.5. The package information is simplified and approximated to generate *simplified package information*, as described in §7.6. The simplified package information is translated into a *control flowgraph* and *structured blocks* as described in §7.7 and §7.8 respectively. The simplified package information and control flowgraph are analysed by the *program analyser algorithm*, controlled by the program analysis heuristics. Guided by the abstract predicates, targeted *candidate invariants* are generated.

The architecture of our program analyser is strongly influenced by the proof planning paradigm. In particular, the program analyser maintains a clear separation between the program analysis framework and the program analysis heuristics. The style means that the program analysis heuristics are presented in a consistent form, facilitating their understanding.

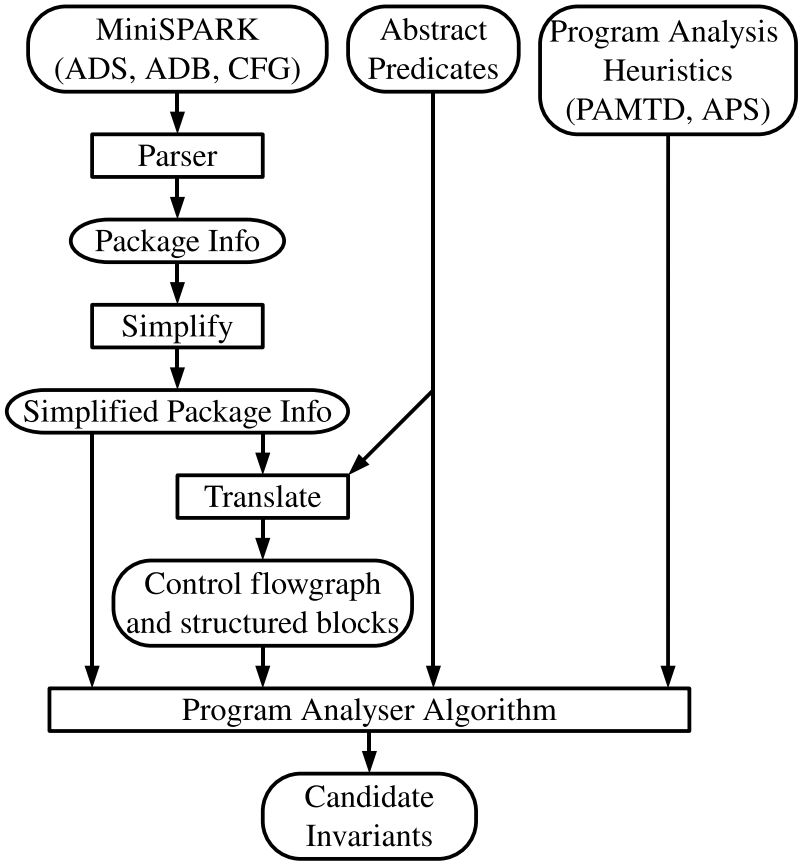


Figure 7.1: Program analysis architecture

7.3 Program Analyser Configuration

The program analyser is configured to support the verification of exception freedom in the SPARK Approach. The program analysis heuristics developed are summarised in §7.10.

7.4 MiniSPARK

Although SPARK is a subset of Ada, it is still a significant programming language. Comprehensive program analysis of SPARK would amount to a substantial implementation effort. Instead, we target a core subset of SPARK as MiniSPARK. The essential behaviour of most imperative programs can be captured though assignment, conditions and loops, all of which are present in MiniSPARK. The complete grammar of MiniSPARK is presented in Appendix F, and is summarised below:

- **Package** - A single package is analysed. The package specification (ADS) declares all types and subprograms while the package body (ADB) contains implementations of the declared subprograms.
- **Target configuration** - As in the SPARK Approach, target specific constraints are provided through a target configuration file (CFG).
- **Types** - Of the scalar types, integer and Boolean are available. Subtypes may be introduced for integer types. Of the composite types, single dimensional arrays are available. No further constraints are imposed, thus arrays of arrays may be constructed.
- **Subprograms and statements** - Both functions and procedures are available. State may be modified directly through assignment or indirectly through procedure calls. Sequencing is supported through if-statements and loops.
- **Expressions** - The arithmetic operators $+$, $-$, $*$, $**$ and *div* are available. The Boolean operations $\wedge$, $\vee$, and $\neg$ are available. The integer relations $=$, $\neq$, $<$, $\leq$, $>$ and $\geq$ are available.

7.5 Parser

The parser transforms MiniSPARK into structures that are more amenable to mechanical processing. Separate structures describe declarations and the program, as described in §7.5.1 and §7.5.2 respectively.

7.5.1 Declarations

As described in §6.5, our proof planner operates in the same logic as the SPARK Approach. Unsurprisingly, there is a strong correspondence between the entities and types

required for program analysis and for program proof. This similarity is exploited by reusing the proof planner declarations for our program analyser. This reuse eases the integration of our program analyser and proof planner. Reasoning problems encountered during program analysis can be directly processed by our proof planner.

7.5.2 Program

The essential semantics of MiniSPARK code is recorded in a collection of structures, as detailed below.

Scalar and Composite Types

The structure for holding scalar and composite types is shown in Figure 7.2 and described below:

- **Integer types** - Each integer type has a declared name *TypeRef*. The type is bounded between the constants *FirstConstantRef* and *LastConstantRef*.
- **Integer subtypes** - Each integer subtype has a declared name *TypeRef*. The parent type or subtype is referenced as *ParentTypeRef*. The subtype is bounded between the constants *FirstConstantRef* and *LastConstantRef*.
- **One dimensional arrays** - Each array has a declared name *TypeRef*. The array is indexed by a scalar type *IndexTypeRef* containing elements of any other type *ElementTypeRef*.

<PackageInfo> ::= *scalarType*(*TypeRef*, *<Scalar>*) |
compositeType(*TypeRef*, *<Composite>*)
<Scalar> ::= *integer*(*FirstConstantRef*, *LastConstantRef*) |
integerSubtype(*ParentTypeRef*, *FirstConstantRef*, *LastConstantRef*)
<Composite> ::= *array*(*IndexTypeRef*, *ElementTypeRef*)

Figure 7.2: Scalar and composite types

Constants

The structure for holding constants is shown in Figure 7.3. Each constant has a name *ConstantRef* and a type *TypeRef*. The corresponding constant expression is recorded as *ConstExp*. Note that only scalar constants are encountered in MiniSPARK.

<PackageInfo> ::= *constant*(*ConstantRef*, *TypeRef*, *ConstExp*)

Figure 7.3: Constants

Subprogram

The structure for holding the declaration of each subprogram is shown in Figure 7.4. Each subprogram has a name *SubprogramRef*, and is identified as being either a procedure or a function. Where the subprogram is a function its return type is recorded as *ReturnTypeRef*.

<*PackageInfo*> ::= *subprogram*(*SubprogramRef*, <*SubprogramKind*>)
<*SubprogramKind*> ::= *procedure* | *function*(*ReturnTypeRef*)

Figure 7.4: Subprograms

Subprogram variables

The structure for holding the variables associated with a subprogram is shown in Figure 7.5 and described below:

- **Parameter** - A subprogram may have zero or more parameter variables. The position of each parameter variable is relevant in handling subprogram calls, and is recorded as *PositionInt*. The variable has name *VarRef*, mode <*VariableMode*> and type *TypeRef*. Three modes are available, describing strictly input parameters (*in*), strictly output parameters (*out*) and input and output parameters (*inout*).

- **Initial parameter variable** - Initial parameter variables reference the value of a variable at the start of a subprogram. An initial parameter variable is available for each parameter variable whose mode includes output. The variable has name *VarRef*~ where its corresponding parameter variable is *VarRef*.

- **Local variable** - A subprogram may have zero or more local variables. Each variable has a name *VarRef* and a type *TypeRef*.

- **For-loop variable** - Each for-loop variable is only in scope for the duration of the loop. The restricted scope is expressed through the subprogram code. The variable has a name *VarRef* and a type *TypeRef*. As standard, the for-loop variable iterates from the first value of its type to the last value of its type. A tighter range, which may potentially be empty, can also be specified, recorded as the initial expression *InitialExp* and final expression *FinalExp*.

- **For-loop entry variable** - The initial and final expressions of a for-loop range are evaluated once, when the loop is entered. To capture these semantics, every variable in a for-loop range is cloned as a special *entry* variable. The expression containing the entry variable is recorded through <*EndPoint*>, as either *initial* or *final*. Each entry variable has a name *VarRef* and is associated with its source variable *CloneVarRef*.

<*PackageInfo*> ::= *subprogramVariable*(*SubprogramRef*, <*VariableDeclare*>)
<*VariableDeclare*> ::= *parameter*(*PositionInt*, *VarRef*,
<*VariableMode*>, *TypeRef*) |
initParameter(*VarRef*~ , *VarRef*) |
localVariable(*VarRef*, *TypeRef*) |
forLoopVariable(*VarRef*, *TypeRef*, <*Range*>) |
forLoopEntryVariable(<*EndPoint*>, *VarRef*, *CloneVarRef*)
<*Range*> ::= *overtype* | *range*(*InitialExp*, *FinalExp*)
<*VariableMode*> ::= *in* | *out* | *inout*
<*EndPoint*> ::= *initial* | *final*

Figure 7.5: Subprogram variables

Subprogram Code

The subprogram code is first normalised into fundamental operations to ease its analysis. The normalisation process is trivial, except for loops. Reflecting the behaviour of the Examiner [Bar03], both while-loops and for-loops are normalised as generic loops as shown in Figure 7.6. While-loops can be expressed directly in terms of generic loops. Additional constructs are required to faithfully reflect the semantics of for-loops. Each for-loop introduces an iterator variable, that is only in scope for the duration of the loop. Explicit scoping constructs are introduced to express the restricted scope. Further, for-loop range expressions are only evaluated when the loop is entered. To express this, variables referenced in for-loop range expressions are cloned as special *entry* variables at loop entry. Range expressions at loop entry may then be accurately expressed in terms of these entry variables. All subprogram annotations are ignored. However, the position of loop invariants are recorded and used during program analysis. The implications of ignoring annotations are explored in greater detail in §9.2.1.

The structure for holding normalised subprogram code is shown in Figure 7.7 and described below:

- **Enter scope** - The variable *VarRef* is now in scope.
- **Exit scope** - The variable *VarRef* is no longer in scope.
- **Assignment** - The data structure referenced through expression *LValueExp* is assigned the result of evaluating expression *RValueExp*.
- **Procedure call** - The procedure named *SubprogramRef* is called with ordered parameter list *ParameterExpList*. Note that functions occur within expressions.
- **Return** - The return statement is only applicable for functions. It identifies the expression returned by a function as *Exp*.
- **If-then** - Where the conditional expression *ConditionExp* evaluates to true the sequence of statements <*TrueStatementList*> are performed.

Structured loop	Normalised loop

while *E* loop
  S;
end loop;
	loop
	  exit when not *E*;
	  S;
	end loop;

for I in T loop
  S;
end loop;
	enterScope(I) {I *of type* T}
	I:=T'First;
	loop
	  S;
	  exit when I=T'Last;
	  I:=T'Succ(I);
	end loop;
	exitScope(I)

for I in T range *L* .. *U* loop
  S;
end loop;
	enterScope(EV^1) ...
	enterScope(EV^n)
	EV^1:=V^1; {V^1 *is in L or U*} ...
	EV^n:=V^n; {V^n *is in L or U*}
	{*EL is L substituting* V^m *with* EV^m}
	{*EU is U substituting* V^m *with* EV^m}
	if *EL* <= *EU* then
	  enterScope(I) {I *of type* T *in EL .. EU*}
	  I:=*EL*;
	  loop
	  S;
	  exit when I=*EU*;
	  I:=T'Succ(I);
	  end loop;
	  exitScope(I)
	end if;
	exitScope(EV^n)
	...
	exitScope(EV^1)

Figure 7.6: Structured loops and their normalised form

- **If-then-else** - Where the conditional expression *ConditionExp* evaluates to true the sequence of statements <*TrueStatementList*> are performed, otherwise the sequence of statements <*FalseStatementList*> are performed.
- **Loop** - Begin a repeated sequence of statements as <*LoopStatementList*>.
- **Mark invariant** - Marks the point within a loop where an invariant is present.
- **If-then-exit** - Where the conditional expression *ConditionExp* evaluates to true the sequence of statements <*TrueStatementList*> are performed and the immediately enclosing loop is exited.

<*PackageInfo*> ::= *subprogramCode*(<*StatementList*>)
<*StatementList*> ::= [] | [<*Statement*> | <*StatementList*>]
<*LoopStatementList*> ::= [] | [<*LoopStatement*> | <*LoopStatementList*>]
<*TrueStatementList*> ::= <*StatementList*>
<*FalseStatementList*> ::= <*StatementList*>
<*Statement*> ::= *enterScope*(*VarRef*) |
exitScope(*VarRef*) |
assign(*LValueExp*, *RValueExp*) |
procedureCall(*SubprogramRef*, *ParameterExpList*) |
return(*Exp*) |
ifThen(*ConditionExp*, <*TrueStatementList*>) |
ifThenElse(*ConditionExp*,
<*TrueStatementList*>, <*FalseStatementList*>) |
loop(<*LoopStatementList*>)
<*LoopStatement*> ::= <*Statement*> |
markInvariant |
ifThenExit(*ConditionExp*, <*TrueStatementList*>)

Figure 7.7: Subprogram code

7.6 Simplifications and Approximations

Before program analysis is conducted it is convenient to simplify and approximate the package information. These transformations are separated from the mechanical operation of the parser as they represent heuristic decisions specific to our analyses.

7.6.1 Replace Named Scalar Constants With Their Values

Constants in the package information may be associated with their corresponding constant expression. Every referenced constant is replaced with the evaluation of its constant expression.

7.6.2 Eliminate Unneeded Casting

Casting must be used to perform arithmetic operations on expressions of different types. However, for convenience, our program analyser is less strict, allowing arithmetic operations between expressions of the same fundamental type. On this basis, all casting is eliminated from the package information.

7.6.3 Transform Return to Assignment

It is convenient to model return statements as an assignment. A local variable is declared as *funret* having the return type of the function. The return statement is then transformed as an assignment to this variable.

7.6.4 Subprogram Call Abstractions

Our program analyser does not recursively analyse called subprograms. Instead, each subprogram call is replaced with an abstraction. An overloaded function is introduced to support subprogram call abstraction:

$$bound(TypeRef) \tag{7.1}$$

The function returns a value, known only to be in type *TypeRef*. Each subprogram declares the mode and type of its parameters. These declarations are exploited to constrain the effect of a subprogram call, as illustrated in Figure 7.8 and described below:

- **Functions** - Functions may have a number of input parameters and will return a single result. Each function is called from within an expression. The function call is abstracted by replacing the function with the appropriate bound function for its result type.
- **Procedures** - Procedures may have a number of input and output parameters. Each procedure is called as a separate statement. The procedure call is abstracted by

replacing the procedure with a sequence of assignment statements, assigning each output parameter the appropriate bound function for its result type.

Function call abstraction
Function declaration: `function` *FunctionRef* `(`*VarRef*^{1} `:` *TypeRef*^{1}, ..., *VarRef*^{n} `:` *TypeRef*^{n} `)` `return` *ReturnTypeRef* `;` Function call from within subprogram: *FunctionRef*`(`*Exp*^{1}, ..., *Exp*^{n}`)` Abstracted function call: *bound*(*ReturnTypeRef*)

Procedure call abstraction
Procedure declaration: `procedure` *ProcedureRef* `(`*VarRef*^{1} `:` *Mode*^{1} *TypeRef*^{1}, ..., *VarRef*^{n} `:` *Mode*^{1} *TypeRef*^{n} `)` `;` Procedure call from within subprogram: *ProcedureRef*`(`*Exp*^{1}, ..., *Exp*^{n}`)` Abstracted procedure call: *assign*(*Exp*^{1}, *bound*(*TypeRef*^{1})) {if *Mode*^{1} is *out* or *inout*} ... *assign*(*Exp*^{n}, *bound*(*TypeRef*^{n})) {if *Mode*^{n} is *out* or *inout*}

Figure 7.8: Subprogram call abstraction

Verifying exception freedom involves proving that variables lie within certain bounds at particular points in a program. In high integrity SPARK programs it is common for subprogram parameters to have tightly constrained types. Thus, where verifying exception freedom, simply abstracting subprogram calls to their result types can provide sufficient constraints. Possible strengthenings of this approach are considered in §9.2.1.

7.7 Control Flowgraph

The subprogram targeted by the abstract predicates is translated into a control flowgraph. A flowgraph is a natural structure for analysing the choice points and actions seen in imperative programing languages. The structure of the flowgraph and its construction from a subprogram is described in §7.7.1 and §7.7.2 respectively.

7.7.1 Control Flowgraph Structure

The structure for holding the control flowgraph is shown in Figure 7.9, and described below:

- **Node** - Nodes describe the choice points and actions seen in a subprogram. Every node has an identifier *NodeId* and contains an item <*NodeItem*>.
- **Edges** - Edges are used to connect nodes together and store properties generated during program analysis. The edge is directed, connecting from node *TailNodeId* to node *HeadNodeId*. Associated with each edge is a property store identifier *PropStoreId*. Multiple properties may be associated with the store, each having an *Address* and a *Property*.

For presentation purposes, a pictorial representation of nodes and edges is introduced in Figure 7.10.

<*FlowGraph*> ::= *node*(*NodeId*, <*NodeItem*>) |
edge(*PropStoreId*, *TailNodeId*, *HeadNodeId*) |
property(*PropStoreId*, *Address*, *Property*)
<*NodeItem*> ::= <*Boundary*> | <*Assignment*> | <*Scope*> | <*Branch*> | <*Merge*>
<*Boundary*> ::= *entry* | *exit*
<*Assignment*> ::= *assign*(*LValueExp*, *RValueExp*)
<*Scope*> ::= *enterScope*(*VarRef*) | *exitScope*(*VarRef*)
<*Branch*> ::= *branch*(*ConditionExp*) | *loopBranch*(*ConditionExp*, *LoopId*)
<*Merge*> ::= *merge* | *loopMerge*(*LoopId*)

Figure 7.9: Control flowgraph framework

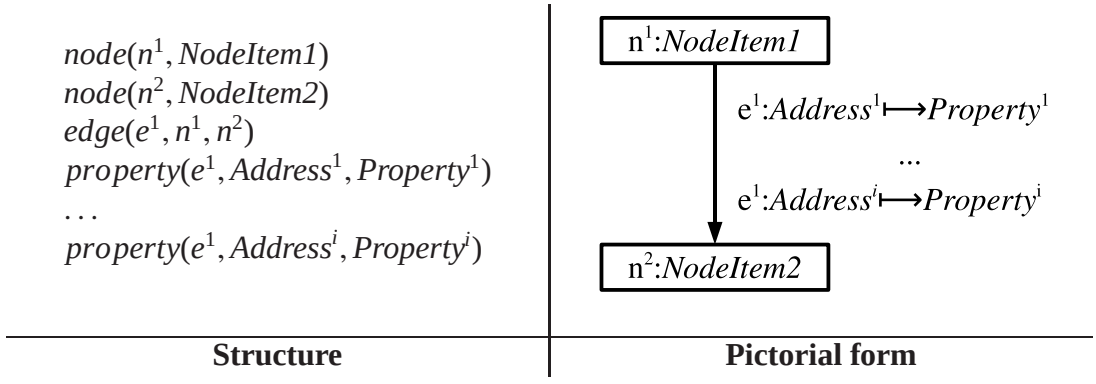


Figure 7.10: Control flowgraph pictorial representation

7.7.2 Subprogram Code as Control Flowgraph

Each component of the subprogram code is expressed through the control flowgraph as detailed below.

Subprogram

The entry and exit points of the subprogram are explicitly recorded, as illustrated in Figure 7.11. Each subprogram has a single entry and exit point.

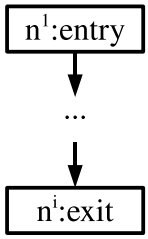


Figure 7.11: Subprogram entry and exit

Assignment Statements

Assignments modify the value of program variables. Following the simplifications and approximations of §7.6, all program variable modifications are expressed in terms of assignment. Each assignment has a single entry and exit point, as illustrated in Figure 7.12.

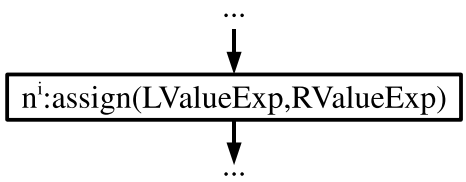


Figure 7.12: Assignment statements

Scope Changes

Scope affects the visibility of program variables. Each scope change has a single entry and exit point. Further, variables entering scope always exit scope on the same path, as illustrated in Figure 7.13.

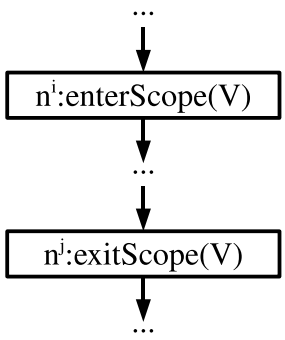


Figure 7.13: Scope changes

Conditional Statements

Conditional statements branch to one of two paths, depending on the truth of a Boolean expression. The two paths will eventually merge, marking the end of the conditional statement. Conditional statements are expressed through branch and merge nodes, as illustrated in Figure 7.14. Properties are attached to the edges leaving the branch node, indicating which path corresponds to which truth value.

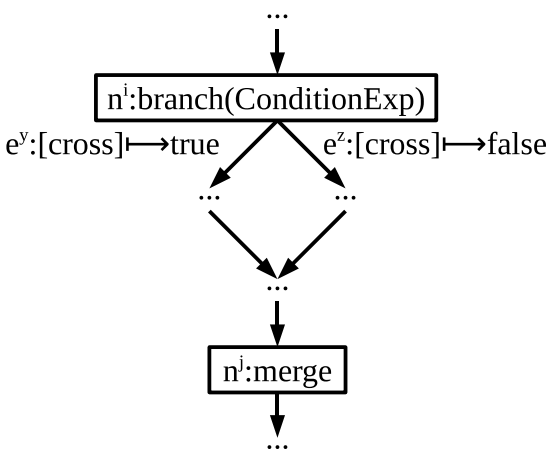


Figure 7.14: Conditional statement

Loop Statements

Loop statements continue to repeat a sequence of statements, depending on the truth of Boolean expressions at loop exit guards. Every path leaving the loop will eventually

merge, marking the end of the loop. Loop statements are expressed through branch and merge nodes, as illustrated in Figure 7.15. Each loop is associated with a unique identifier, grouping its corresponding branch and merge nodes. Properties are attached to the edges leaving branch nodes, indicating which path corresponds to which truth value. Further, a property is attached to the edge that corresponds to the location of the loop invariant.

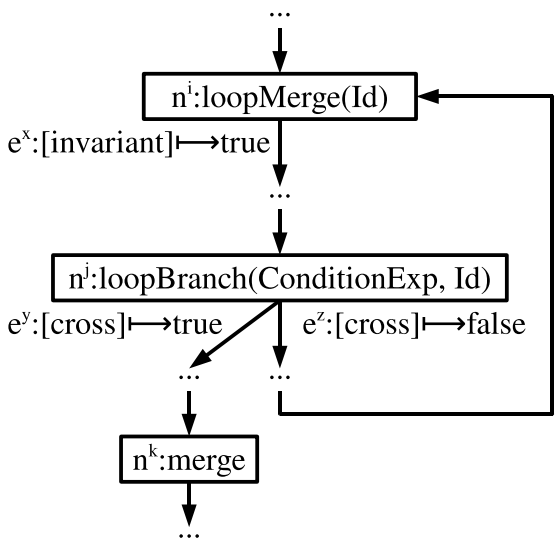


Figure 7.15: Loop statement

7.8 Structured Blocks

The program analysis heuristics involve traversing routes through the control flowgraph. It is convenient to express these routes in terms of the main structured blocks encountered. Each structured block has a single entry and exit point. The block may contain paths, which are described as a sequence of structured blocks. The structure for holding the structured blocks is shown in §7.16, and detailed below:

- *unit*(*NodeId*) -
  The unit block describes a boundary, assignment or scope statement, at *NodeId*.

- *test*(*BranchNodeId*, <*TruePath*>, <*FalsePath*>, *MergeNodeId*) -
  The test block describes a conditional statement. The test block is entered at the branching node *BranchNodeId*. The true and false paths are <*TruePath*> and <*FalsePath*> respectively. The test block is exited where these two paths merge at *MergeNodeId*.

- *loop*(*LoopMergeNodeId*, <*Path*>, *MergeNodeId*) -
  The loop block describes a loop statement. The loop block is entered at the loop merge node *LoopMergeNodeId*. The path around the loop is <*Path*>, which will

contain at least one loop test block. The loop block is exited where all of the paths exiting the loop merge at *MergeNodeId*.

- *looptest*(*BranchNodeId*, <*TruePath*>) -
  The loop test block describes a conditional statement which, if true, will lead to control breaking out of a loop. These blocks can only occur on the paths around a loop. The loop test block is entered at the branching node *BranchNodeId*. The false path is implicitly covered by the enclosing loop, while the true path is <*TruePath*>. The loop test block is exited at the end of its true path, just prior to reaching the loop merge node.

- *subprogram*(*EntryNodeId*, <*Path*>, *ExitNodeId*) -
  The subprogram block describes the entire subprogram. The subprogram block is entered at the subprogram entry node *EntryNodeId*. The path through the subprogram is <*Path*>. The subprogram block is exited at the subprogram exit node *ExitNodeId*.

<*Block*> ::= *unit*(*NodeId*) |
test(*BranchNodeId*, <*TruePath*>, <*FalsePath*>, *MergeNodeId*) |
loop(*LoopMergeNodeId*, <*Path*>, *MergeNodeId*) |
looptest(*BranchNodeId*, <*TruePath*>) |
subprogram(*EntryNodeId*, <*Path*>, *ExitNodeId*)
<*BlockList*> ::= [] | [*Block* | <*BlockList*>]
<*Path*> ::= <*BlockList*>
<*TruePath*> ::= <*Path*>
<*FalsePath*> ::= <*Path*>

Figure 7.16: Structured blocks

7.9 Program Analyser Algorithm

The program analyser algorithm performs the program analysis. The target subprogram is described through the simplified package information, control flowgraph and structured blocks. The analysis of the subprogram is performed by program analysis heuristics. Program analysis methods discover relevant program properties, as described in §7.9.1. With these properties in place, abstract predicate satisfiers suggest targeted invariant strengthening, as described in §7.9.2.

7.9.1 Program Analysis Methods

The program analysis is performed by program analysis methods. Each *program analysis method* (PAMTD) is expressed through three features, as described below:

- **Property type** - The method attempts to discover properties that hold at the edges of the control flowgraph. The *property type* defines the type of these properties. Each property type declares an address and the structure of the values it holds, as shown in Figure 7.17.

- **Route** - The method traverses the flowgraph following a *route*. The route need not visit every node in the flowgraph. Further, the route may be modified during analysis.

- **Property operations** - The method propagates properties along its selected route via *property operations*. A property operation is defined for every node that may be encountered on the selected route. Typically, the property operation describes how the properties arriving at a node are transformed by the node. The property operation may exploit the simplified package information and properties discovered by other methods.

The intention is that the program analysis methods will produce correct results. However, this is not explicitly verified. Thus, the program analyser is regarded as generating *candidate invariants*. The correctness of all selected invariants is demonstrated during program verification.

Address $\mapsto$ Property
$[Item_1, \ldots Item_n] \mapsto$ <*Property*>
Definitions
<*Property*> ::= *Structure*

Figure 7.17: Property type

7.9.2 Abstract Predicate Satisfiers

Abstract predicates request targeted invariant strengthening. Each abstract predicate is associated with an *abstract predicate satisfier* (APS). The satisfier describes how to exploit the available properties to generate the invariant strengthening requested.

7.10 Program Analysis Heuristics Overview

Program analysis heuristics are developed to support the verification of exception freedom in the SPARK Approach. An overview of the program analysis methods are given in §7.10.1, and detailed in Appendix G. The abstract predicate satisfiers are defined in §7.10.2.

7.10.1 Program Analysis Methods

Each program analysis method discovers a type of program properties. The program analysis methods are ordered, allowing methods to exploit properties discovered by earlier methods. The methods are summarised below, in their order of application. The first methods discover richer information about the subprogram under analysis.

- scope (§G.2) - Discovers which variables are in scope.
- update (§G.3) - Discovers which variables have been fully assigned and where these assignments may have taken place.
- context (§G.4) - Discovers the structural contexts that exist within the subprogram. Each structural context corresponds to taking a particular path through the subprogram.

The next methods discover simple, constraint based, properties. These properties may be expressed as invariants to support the verification of exception freedom.

- type (§G.5) - Discovers constraints for all variables, based on their declared type.
- transient (§G.6) - Discovers properties that hold for sections of the subprogram. Following each conditional statement, a property may be introduced indicating that the statement, or its negation, holds. Further, the property continues to hold while its variables are not updated and the structural context remains the same.
- loop_range (§G.7) - A for-loop variable may have a corresponding range constraint. These constraints are identified and added as properties.

The final method performs a richer analysis, to discover invariant constraints. These properties are occasionally needed to support the verification of exception freedom.

- int_constraint (§G.8) - Discovers invariant constraints for integer variables within loops, through the construction and solving of recurrence relations.

7.10.2 Abstract Predicate Satisfiers

The abstract predicate satisfiers describe how to fulfil an abstract predicate by exploiting properties discovered by the program analysis methods. Each abstract predicate has a corresponding abstract predicate satisfier as listed below:

- *abstractPredicate*(*SubprogramName*, *constrainVars*(*VarList*)) -
  Candidate invariants are sought that constrain the variables *VarList* in subprogram *SubprogramName.* The type method is queried at invariant points to introduce type constraints for each variable.

- *abstractPredicate*(*SubprogramName*, *tightlyConstrainVars*(*VarList*)) -
  Candidate invariants are sought that tightly constrain the variables *VarList* in subprogram *SubprogramName.* The loop_range, transient and int_constraint methods are queried to introduce all constraints that reference the targeted variables. Constraints that reference entry variables are omitted, as these are constrained indirectly.

- *abstractPredicate*(*SubprogramName*, *coupleWithEntryVars*(*VarList*)) -
  Candidate invariants are sought that couple variables *VarList* with their corresponding entry variables in subprogram *SubprogramName.* The transient method is queried to introduce constraints that relate a targeted variable to its corresponding entry variable.

Note that a candidate invariant is rejected if it is already present as a program invariant[1].

[1] In practice, as described in §7.5.2, our program analyser only records the location of invariants, not their expressions. Thus, the rejection of candidate invariants that would duplicate existing program invariants is preformed manually.

Chapter 8

Evaluation

8.1 Introduction

In this chapter SPADEase is evaluated. The implementation of SPADEase is described in §8.2. The evaluation process is described in §8.3. Finally, both textbook and industrial subprograms are evaluated in §8.4 and §8.5 respectively. An overall analysis of these results is presented in §8.6.

8.2 Implementation of SPADEase

As detailed in Chapter 5, SPADEase contains both a proof planner and a program analyser. The implementation of these components are considered below.

8.2.1 Implementing the Proof Planner

As observed in §2.3.3, there are number of existing proof planner systems. The critics enabled version of the Clam proof planner has previously been used to support program verification. On this basis, the planner was used to support our initial investigations. Targeted modifications were made such that Clam could import goals from the SPARK Approach and export discovered proof plans to the Checker.

While this prototype supported our initial investigations, the implementation had a number of weaknesses. The integration with the SPARK Approach was limited, requiring custom configurations in planning each goal. The goal representation did not support the storage of contextual information. The planner algorithm was recursive, hindering the development of global analysis critics. Finally, the planner offered little support for analysing the progress of a proof plan.

The various limitations of Clam were addressed through the development of our own proof planner. Our proof planner was developed in Sicstus Prolog. The method-language includes the clp(FD) constraint solver [COC97], as distributed with Sicstus Prolog. Further, while direct communication has not been implemented, the method-language is

supported through the computer algebra system YACAS [YAC]. Finally, the method-language present in Clam was included, primarily to reuse its expression analysis and rippling predicates.

8.2.2 Implementing the Program Analyser

As detailed in §7.4, developing a program analyser for the entire SPARK language would represent a significant undertaking. Thus, we restrict our analysis to a core subset of SPARK as MiniSPARK. To provide an effective integration with our proof planner we developed our own program analyser, reusing existing technologies where available. Praxis supplied a complete SPARK grammar and tokeniser. Building on these components, Stratego [Vis01] was used to translate MiniSPARK programs into analysable structures. The program analyser itself is implemented in Sicstus Prolog. As observed in §7.5.1, our program analyser reuses declarations and associated functionality from our proof planner to offer a more effective integration. While direct communication has not been implemented, a method is supported through the recurrence relation solver PURRS [PUR].

8.3 Evaluation of SPADEase

As described in Chapter 5, SPADEase attempts to enhance the verification of exception freedom in the SPARK Approach. As observed in §4.4.4, program verification in the SPARK Approach is decompositional. Reflecting this decomposition, SPADEase is evaluated on individual subprograms. As highlighted in §2.2, program verification involves both proof discovery and invariant discovery. To capture the complete verification process, the selected subprograms initially contain only default invariants. As described in §5.3, the verification of exception freedom in the SPARK Approach, as enhanced with SPADEase, involves an iterative process. A concise result format is introduced in §8.3.1 to describe each iteration. Supplementary text is added wherever SPADEase requires manual interaction or leaves unproven goals.

8.3.1 Result Format

The following table is used in describing an application of the SPARK Approach as enhanced with SPADEase:

Cyclomatic complexity: *s*, Max loop vars: *t*, Max loop arith ops: *u*															
Goals	**Form**	**Strategy**				**Critic**					**Prog**				
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration *v*: *w* initial goals, *x* remaining goals, *y*s to simplify, *z*s to plan															
abstract predicates															
invariants															

The complexity of the subprogram is indicated through three metrics. The *cyclomatic complexity* [McC76] reports on the path complexity of the subprogram. The *maximum loop variables* reports the maximum number of variables, as defined by prog_var_exps, encountered inside a loop. The *maximum loop arithmetic operators* reports the maximum number of distinct arithmetic operators encountered inside a loop.

Each iteration begins with an application of the SPARK Approach. The Examiner analyses the target subprogram, generating the initial goals. The Simplifier attempts to automatically prove these goals, returning any remaining goals. As the SPARK Approach is documented elsewhere [AC02, BCJ+06], its details are omitted. Instead, the number of initial and remaining goals are reported. The simplification time and planning time for each iteration is reported in seconds, to give an indication of both complexity and performance. Note that the evaluation was performed on an NC10 netbook, with a Atom N270 processor, 1GB of RAM and the Ubuntu 9.10 operating system. The essential behaviour of SPADEase is described for each remaining goal accepted by the targeted_goal method as follows:

- **Goals** - Number of goals with the same characteristics.
- **Form** - Run-time check (*rtc*), returning invariant (*rinv*) or between invariant (*binv*) goals.
- **Strategy**, **Critic** and **Prog** - Aliases defined in table below:

Strategy	**ef**	exception_freedom	§E.3
	rc	run_time_check	§E.4
	tr	transitivity	§E.4
	ri	ripple	§E.5
Critic	**cc**	constrain_consts	§E.15
	ce	couple_entry_vars	§E.14
	cv	constrain_vars	§E.16
	tc	tightly_constrain_vars	§E.17
	fc	false_conc	§E.21
Prog (Program Analysis Heuristics)	**ty**	type	§G.5
	lr	loop_range	§G.7
	ts	transient	§G.6
	ic	int_constraint	§G.8

The symbol ● denotes that a feature was invoked, and succeeded. The symbol ○ denotes that a feature was invoked and failed, however the failure triggered successful failure analysis. Finally, the symbol ⊗ denotes that a feature was invoked and failed completely.

At the end of each iteration, SPADEase may satisfy abstract predicates through invariant strengthening. For the non-industrial examples, the form of the abstract predicates and their corresponding strengthened invariants are shown.

8.4 Textbook Subprograms

8.4.1 Subprogram Average

The initially annotated Average subprogram is shown in Figure 8.1. The subprogram reports the mean average value stored in an array.

```
package Average_Package
is
  subtype AR_T is Integer range 10..100;
  subtype AE_T is Integer range 0..10;
  subtype SumRange is Integer range AE_T'First*((AR_T'Last-AR_T'First)+1) ..
                                    AE_T'Last*((AR_T'Last-AR_T'First)+1);
  type A_T is array (AR_T) of AE_T;
  procedure Average(D: in A_T; A: out AE_T);
  --# derives A from D;
end Average_Package;
```

```
package body Average_Package
is
  procedure Average(D: in  A_T; A: out AE_T)
  is
    S: SumRange;
  begin
    S:=0;
    for I in AR_T loop
      --# assert true;
      S:=S+D(I);
    end loop;
    A:=S/((AR_T'Last-AR_T'First)+1);
  end Average;
end Average_Package;
```

Figure 8.1: Average subprogram

Cyclomatic complexity: 2, Max loop vars: 3, Max loop arith ops: 1															
Goals	**Form**	**Strategy**				**Critic**					**Prog**				
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 23 initial goals, 1 remaining goals, 1s to simplify, 7s to plan															
1	*rtc*	○							●					●	
(prog analysis)	*tightlyConstrainVars*([*d*, *i*, *s*])														
(sole loop)	$(s \geq 0) \wedge (s \leq (i - 10) * 10)$														
Iteration 2: 25 initial goals, 2 remaining goals, 2s to simplify, 10s to plan															
1	*rinv*	●	●		●										
1	*rtc*	●	●	●											

8.4.2 Subprogram BubbleSort

The initially annotated BubbleSort subprogram is shown in Figure 8.2. The subprogram sorts an array using the bubble sort algorithm.

```
package BubbleSort_Package is
  subtype AR_T is Integer range 1..10;
  type A_T is array (AR_T) of Integer;
  procedure BubbleSort(A: in out A_T);
  --# derives A from A;
end BubbleSort_Package;
```

```
package body BubbleSort_Package is
  procedure BubbleSort(A: in out A_T)
  is
    Tmp: Integer;
  begin
    for I in AR_T range AR_T'First..(AR_T'Last-AR_T'First) loop
      --# assert true;
      for J in AR_T range AR_T'First..(AR_T'Last-I) loop
        --# assert true;
        if A(J)>A(J+1) then
          Tmp:=A(J);
          A(J):=A(J+1);
          A(J+1):=Tmp;
        end if;
      end loop;
    end loop;
  end BubbleSort;
end BubbleSort_Package;
```

Figure 8.2: BubbleSort subprogram

Cyclomatic complexity: 6, Max loop vars: 4, Max loop arith ops: 2

Goals	Form	Strategy				Critic					Prog			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 84 initial goals, 1 remaining goals, 3s to simplify, 7s to plan														
1	*rtc*	○					●						●	
(prog analysis)	*coupleWithEntryVars*([*i*])													
(inner loop)	*i* = *i_entry*													
Iteration 2: 84 initial goals, 0 remaining goals, 2s to simplify, 6s to plan														

8.4.3 Subprogram DualFilter

The initially annotated DualFilter subprogram is shown in Figure 8.3. The subprogram sums the multiple of all elements from two arrays that lie within constant bounds.

```
package DualFilter_Package is
  subtype AR_T is Integer range 0..9;
  subtype AE_T is Integer range -200..1000;
  type A_T is array (AR_T) of AE_T;
  procedure DualFilter(D1: in A_T; D2: in A_T; P: out Integer);
  --# derives P from D1, D2;
end DualFilter_Package;
```

```
package body DualFilter_Package is
  procedure DualFilter(D1: in A_T; D2: in A_T; P: out Integer)
  is
  begin
    P:=0;
    for I in AR_T loop
      --# assert true;
      if D1(I) >=-100 and D1(I)<=-50 and
         D2(I) >=300 and D2(I)<=900 then
        P:=P+(D1(I)*D2(I));
      end if;
    end loop;
  end DualFilter;
end DualFilter_Package;
```

Figure 8.3: DualFilter subprogram

Cyclomatic complexity: 3, Max loop vars: 4, Max loop arith ops: 2															
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 40 initial goals, 5 remaining goals, 1s to simplify, 19s to plan														
5	*rtc*	○				●								
(interaction)	*constrainConsts*([*system_min_int*, *system_max_int*])													
Iteration 2: 40 initial goals, 1 remaining goals, 1s to simplify, 11s to plan														
1	*rtc*	○								●				
(prog analysis)	*tightlyConstrainVars*([*p*, *element*(*d1*, [*i*]), *element*(*d2*, [*i*])])													
(sole loop)	$(p \geq (i * -90000)) \wedge (p \leq 0)$													
Iteration 3: 43 initial goals, 3 remaining goals, 3s to simplify, 42s to plan														
1	*rinv*	●	●	●	●									
1	*rinv*	●	●		●									
1	*rtc*	●	●											

In iteration 2 SPADEase requests an engineer to introduce constraints for undefined constants. The constants are required to describe the bounds of numeric literals on the target architecture. They arise in this subprogram due to the presence of constant expressions in the package body. In response, appropriate constraints are introduced by manually extending the target configuration file associated with the subprogram.

8.4.4 Subprogram MatrixFilter

The initially annotated MatrixFilter subprogram is shown in Figure 8.4. The subprogram sums the elements from a two dimensional array that lie within a subtype.

```
package MatrixFilter_Package is
  subtype I_T is Integer range 0 .. 10;
  subtype E_T is Integer range 0 .. 500;
  subtype F_T is Integer range 100 .. 200;
  subtype R_T is Integer range
    0..F_T'Last*(((I_T'Last-I_T'First)+1)**2);
  type InOne_T is array (I_T) of Integer;
  type InTwo_T is array (I_T) of InOne_T;
  procedure MatrixFilter(A: in InTwo_T;
                         R: out R_T);
  --# derives R from A;
end MatrixFilter_Package;
```

```
package body MatrixFilter_Package is
  procedure MatrixFilter(A: in InTwo_T;
                         R: out R_T)
  is
  begin
    R:=0;
    for I in I_T loop
      --# assert true;
      for J in I_T loop
        --# assert true;
        if A(I)(J)>=F_T'First and A(I)(J)<=F_T'Last then
          R:=R+A(I)(J);
        end if;
      end loop;
    end loop;
  end MatrixFilter;
end MatrixFilter_Package;
```

Figure 8.4: MatrixFilter subprogram

Cyclomatic complexity: 4, Max loop vars: 4, Max loop arith ops: 1															
Goals	**Form**	**Strategy**				**Critic**					**Prog**				
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 55 initial goals, 1 remaining goals, 1s to simplify, 7s to plan															
1	*rtc*	○								●				●	
(prog analysis)		*tightlyConstrainVars*([*element*(*element*(*a*, [*i*]), [*j*]), *r*])													
(inner loop)		$(r \geq 0) \wedge (r \leq ((i * 2200) + (j * 200)))$													
Iteration 2: 58 initial goals, 5 remaining goals, 7s to simplify, 29s to plan															
2	*binv*	○							●		●				
2	*rinv*	●	●		●										
1	*rinv*	●	●	●											
(prog analysis)		*constrainVars*([*r*])													
(outer loop)		$(r \geq 0) \wedge (r \leq 24200)$													
(inner loop)		$(r \geq 0) \wedge (r \leq ((i * 2200) + (j * 200)))$													
Iteration 3: 61 initial goals, 4 remaining goals, 9s to simplify, 27s to plan															
1	*rtc*	○								●				●	
2	*rinv*	●	●		●										
1	*rinv*	●	●	●											
(prog analysis)		*tightlyConstrainVars*([*r*, *i*])													
(outer loop)		$((r \geq 0) \wedge (r \leq 24200)) \wedge ((r \geq 0) \wedge (r \leq (i * 2200)))$													
(inner loop)		$(r \geq 0) \wedge (r \leq ((i * 2200) + (j * 200)))$													
Iteration 4: 67 initial goals, 5 remaining goals, 32s to simplify, 73s to plan															
2	*binv*	○	○	○						●				⊗	
2	*rinv*	●	●		●										
1	*rtc*	●	●	●											

In iteration 3, proof failure analysis requests the introduction of tighter constraints for variables *r* and *i*. Through program analysis, tighter constraints are discovered for *r* and are introduced through a strengthened invariant. In iteration 4, with the strengthened invariant in place, every goal is provable. However, despite this, proof failure analysis requests the introduction of tighter constraints for the same variables *r* and *i* and also for variables *a* and *j*. This flawed proof failure analysis occurs as the transitivity strategy fails to prove two provable goals. The key problem is that, as the proof is developed, multiple occurrences of variable *i* emerge. The transitivity strategy treats each occurrence independently, leading to the introduction of weaker constraints and the failure of the proof.

8.4.5 Subprogram MatrixMult

The initially annotated MatrixMult subprogram is shown in Figure 8.5. The subprogram performs matrix multiplication.

```
package MatrixMult_Package is
  subtype I_T is Integer range 0 .. 3;
  subtype E_T is Integer range -9 .. 9;
  subtype R_T is Integer range
    ((E_T'First*E_T'Last)*((I_T'Last-I_T'First)+1))..
    ((E_T'Last*E_T'Last)*((I_T'Last-I_T'First)+1));
  type InOne_T is array (I_T) of E_T;
  type InTwo_T is array (I_T) of InOne_T;
  type OutOne_T is array (I_T) of R_T;
  type OutTwo_T is array (I_T) of OutOne_T;
  procedure InitToZero(R: out OutTwo_T);
  --# derives R from ;
  procedure MatrixMult(A: in InTwo_T; B: in InTwo_T; R: out OutTwo_T);
  --# derives R from A, B;
end MatrixMult_Package;
```

```
package body MatrixMult_Package is
  procedure InitToZero(R: out OutTwo_T)
  is
  begin
    for I in I_T loop
      for J in I_T loop
        R(I)(J):=OutTwo_T'First;
      end loop;
    end loop;
  end InitToZero;
  procedure MatrixMult(A: in InTwo_T; B: in InTwo_T; R: out OutTwo_T)
  is
    M: Integer;
  begin
    InitToZero(R);
    for I in I_T loop
      --# assert true;
      for J in I_T loop
        --# assert true;
        M:=0;
        for K in I_T loop
          --# assert true;
          M:=M+A(I)(K)*B(K)(J);
        end loop;
        R(I)(J):=M;
      end loop;
    end loop;
  end MatrixMult;
end MatrixMult_Package;
```

Figure 8.5: MatrixMult subprogram

Cyclomatic complexity: 4, Max loop vars: 7, Max loop arith ops: 2														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 102 initial goals, 4 remaining goals, 2s to simplify, 43s to plan														
4	*rtc*	○	○	○						●				●
(prog analysis)	*tightlyConstrainVars*([*m*, *element*(*element*(*a*, [*i*]), [*k*]), *element*(*element*(*b*, [*k*]), [*j*])])													
(inner-most loop)	$(m \geq (k * -81)) \wedge (m \leq (k * 81))$													
Iteration 2: 104 initial goals, 3 remaining goals, 5s to simplify, 121s to plan														
1	*rinv*	●	●	●	●									
2	*rtc*	●	●	●										

8.4.6 Subprogram OpenPortScan

The initially annotated OpenPortScan subprogram is shown in Figure 8.6. The subprogram counts the number of open ports within a provided range. The range is expressed through a starting port and a number of subsequent ports.

```
package OpenPortScan_Package is
  subtype PortRange is Integer range 0..(2**16)-1;
  subtype PortTotal is Integer range 0..(PortRange'Last-PortRange'First)+1;
  type Ports is array (PortRange) of Boolean;
  function PortIsOpen(StatusOfPorts: in Ports;
                      Port:          in PortRange) return Boolean;
  procedure OpenPortScan(StatusOfPorts: in  Ports;
                         PStart:        in  PortRange;
                         PNum:          in  PortRange;
                         TOpen:         out PortTotal;
                         Error:         out Boolean);
  --#derives TOpen, Error from StatusOfPorts, PStart, PNum;
end OpenPortScan_Package;
```

```
package body OpenPortScan_Package is
  function PortIsOpen(StatusOfPorts: in Ports;
                      Port:          in PortRange) return Boolean
  is
  begin
    return StatusOfPorts(Port);
  end PortIsOpen;
  procedure OpenPortScan(StatusOfPorts: in  Ports;
                         PStart:        in  PortRange;
                         PNum:          in  PortRange;
                         TOpen:         out PortTotal;
                         Error:         out Boolean)
  is
  begin
    Error:=False;
    TOpen:=0;
    if ((PStart+PNum)<=PortRange'Last) then
      for I in PortRange range PStart..(PStart+PNum) loop
        --# assert true;
        if (PortIsOpen(StatusOfPorts, I)) then
          TOpen:=TOpen+1;
        end if;
      end loop;
    else
      Error:=True;
    end if;
  end OpenPortScan;
end OpenPortScan_Package;
```

Figure 8.6: OpenPortScan subprogram

Cyclomatic complexity: 5, Max loop vars: 5, Max loop arith ops: 1

Goals	Form	Strategy				Critic					Prog			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 45 initial goals, 3 remaining goals, 1s to simplify, 5s to plan														
2	*rinv*	○					●						●	
1	*rtc*	○					●						●	
(prog analysis)	*coupleWithEntryVars*([*pstart*, *pnum*])													
(sole loop)	(*pstart* = *pstart_entry*) ∧ (*pnum* = *pnum_entry*													
Iteration 2: 48 initial goals, 3 remaining goals, 2s to simplify, 11s to plan														
2	*rinv*	○							●			●	●	●
2	*rtc*	○							●			●	●	●
(prog analysis)	*tightlyConstrainVars*([*topen*, *i*, *pnum*, *pstart*])													
(sole loop)	(*i* ≥ *pstart_entry*) ∧ (*i* ≤ (*pstart_entry* + *pnum_entry*)) ∧ ((*pstart* + *pnum*) ≤ 65535) ∧ (*topen* ≥ 0) ∧ (*topen* ≤ (*i* − *pstart_entry*))													
Iteration 3: 63 initial goals, 0 remaining goals, 3s to simplify, 4s to plan														

8.4.7 Subprogram ResetArray

The initially annotated ResetArray subprogram is shown in Figure 8.7. The subprogram resets all elements of an array to a provided integer value, so long as this value is within the element type of the array.

```
package ResetArray_Package
is
  subtype AR_T is Integer range 0..100;
  subtype AE_T is Integer range 0..10;
  type A_T is array (AR_T) of AE_T;
  procedure ResetArray(V: in Integer; A: in out A_T);
  --# derives A from V, A;
end ResetArray_Package;
```

```
package body ResetArray_Package
is
  procedure ResetArray(V: in Integer; A: in out A_T)
  is
  begin
    if (V>=AE_T'First and V<=AE_T'Last) then
      for I in AR_T loop
        --# assert true;
        A(I):=V;
      end loop;
    end if;
  end ResetArray;
end ResetArray_Package;
```

Figure 8.7: ResetArray subprogram

Cyclomatic complexity: 3, Max loop vars: 3, Max loop arith ops: 0															
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 45 initial goals, 3 remaining goals, 1s to simplify, 3s to plan														
2	*rtc*	○	○										●	
(prog analysis)	*tightlyConstrainVars*([*v*])													
(sole loop)	$(v \geq 0) \wedge (v \leq 10)$													
Iteration 2: 20 initial goals, 0 remaining goals, 1s to simplify, 2s to plan														

8.5 Industrial Subprograms

SPADEase is evaluated against two industrial applications. These applications include the Ship Helicopter Operating Limits Information System (SHOLIS) [KHCP00]. Each application has roughly fifteen thousand lines of code. In verifying exception freedom, each application leads to roughly seven thousand VCs. The applications have been verified as being free from exceptions. Further, the design and implementation of the applications took this verification task into consideration.

Our attention is focused on loop-based code that is not automatically verified by the SPARK Approach. On this basis, the industrial subprograms were collected via the following procedure:

- **Remove all loop invariants** - The industrial applications have sufficient invariants to verify exception freedom. These invariants are removed to reflect the genuine verification effort.
- **Apply the SPARK Approach** - The SPARK Approach is applied to generate the initial and remaining VCs for each subprogram.
- **Collect loop-based subprograms with remaining VCs** - Each subprogram with at least one loop and some remaining VCs is investigated. Subprograms entirely written in SPARK, and not associated with control loops, are collected for evaluation.

Recall from Chapter 7 that our program analyser operates on a restricted subset of SPARK as MiniSPARK. For this reason, the program analysis aspect of the industrial evaluation was achieved by manually simulating the program analysis heuristics.

8.5.1 Subprogram 1

Cyclomatic complexity: 3, Max loop vars: 3, Max loop arith ops: 0

Goals	Form	Strategy				Critic					Prog			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 38 initial goals, 2 remaining goals, 3s to simplify, 5s to plan														
1	*rinv*	○					●						●	
1	*rtc*	○					●						●	
Iteration 2: 39 initial goals, 2 remaining goals, 2s to simplify, 8s to plan														
1	*rinv*	○						●			●			
1	*rtc*	○						●			●			
Iteration 3: 41 initial goals, 1 remaining goals, 2s to simplify, 7s to plan														
1	*rtc*	○							●			●		
Iteration 4: 43 initial goals, 0 remaining goals, 2s to simplify, 3s to plan														

8.5.2 Subprogram 2

Cyclomatic complexity: 5, Max loop vars: 5, Max loop arith ops: 1															
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 127 initial goals, 11 remaining goals, 19s to simplify, 46s to plan														
11	*rtc*	○					●						●	
Iteration 2: 135 initial goals, 0 remaining goals, 17s to simplify, 29s to plan														

8.5.3 Subprogram 3

Cyclomatic complexity: 4, Max loop vars: 8, Max loop arith ops: 3														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 69 initial goals, 5 remaining goals, 19s to simplify, 20s to plan														
2	*rinv*	○					●						●	
3	*rtc*	○					●						●	
Iteration 2: 75 initial goals, 3 remaining goals, 20s to simplify, 18s to plan														
1	*rtc*	○							●			●		
2	*rtc*	○							●					●
Iteration 3: 87 initial goals, 0 remaining goals, 19s to simplify, 7s to plan														

8.5.4 Subprograms 4 and 5

Cyclomatic complexity: 4, Max loop vars: 4, Max loop arith ops: 1														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 86 initial goals, 4 remaining goals, 6s to simplify, 110s to plan														
4	*rtc*	○				●								
Iteration 2: 86 initial goals, 4 remaining goals, 6s to simplify, 89s to plan														
2	*rtc*	○	○	○						●				●
2	*rtc*	●	●	●										
Iteration 3: 90 initial goals, 6 remaining goals, 12s to simplify, 243s to plan														
2	*rinv*	●	●	●	●									
4	*rtc*	●	●	●										

Two subprograms of similar functionality produced exactly the same results. At iteration 1, SPADEase requests an engineer to constrain a constant array. In response, appropriate user rules are manually introduced to constrain the array.

8.5.5 Subprogram 6

Cyclomatic complexity: 4, Max loop vars: 13, Max loop arith ops: 3															
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 182 initial goals, 9 remaining goals, 61s to simplify, 107s to plan														
2	*rinv*	○					●						●	
7	*rtc*	○					●						●	
Iteration 2: 185 initial goals, 9 remaining goals, 65s to simplify, 276s to plan														
2	*rinv*	○						●			●			
7	*rtc*	○						●			●			
Iteration 3: 191 initial goals, 7 remaining goals, 65s to simplify, 113s to plan														
3	*rtc*	○	○	○						●				●
2	*rtc*	○	○							●		●		
2	*rtc*	⊗	⊗	⊗										
Iteration 4: 203 initial goals, 5 remaining goals, 706s to simplify, 198s to plan														
1	*rinv*	●	●	●	●									
1	*rinv*	●	●		●									
3	*rtc*	●	●	●										

In iteration 3, SPADEase fails to make any progress for two goals. However, for other goals in the same iteration, proof failure analysis triggers the introduction of stronger invariants. In iteration 4, with these stronger invariants in place, the two goals are proved by the Simplifier.

8.5.6 Subprogram 7

Cyclomatic complexity: 9, Max loop vars: 7, Max loop arith ops: 1															
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 626 initial goals, 25 remaining goals, 28s to simplify, 335s to plan														
14	*rtc*	○					●						●	
Iteration 2: 626 initial goals, 23 remaining goals, 27s to simplify, 390s to plan														
12	*rtc*	○						●			●			
Iteration 3: 644 initial goals, 12 remaining goals, 24s to simplify, 323s to plan														
2	*rtc*	○	○	○						●				●
Iteration 4: 650 initial goals, 11 remaining goals, 26s to simplify, 343s to plan														
2	*rtc*	○	○	○						●		●		
Iteration 5: 656 initial goals, 11 remaining goals, 29s to simplify, 352s to plan														
2	*rtc*	○	○	○						●				●
Iteration 6: 662 initial goals, 9 remaining goals, 29s to simplify, 361s to plan														
2	*rtc*	○	○	○						●			⊗	⊗

In iteration 5, proof failure analysis requests the introduction of tighter constraints for selected variables. Through program analysis, tighter constraints are discovered and

introduced through a strengthened invariant. However, in iteration 6, proof failure analysis requests the introduction of tighter constraints for exactly the same variables. The program analyser is unable to deliver tighter constraints, causing SPADEase to make no more progress on these goals. From inspecting the code, a key equality constraint is not discovered by the program analyser. Generalisations made in the int_constraint method, to conform to its property type, prevent the equality constraint from being discovered. If manually introducing the equality constraint, neither the Simplifier nor SPADEase can automatically prove the goals. The key step in the proof involves exploiting a contradiction among the hypotheses.

In each iteration, goals are rejected by the targeted_goal method. These goals all relate to verifying preconditions. While SPADEase does not consider these goals directly, invariants introduced to advance the verification of exception freedom lead to the number of these unconsidered remaining goals, 0 seconds falling from 9 in iteration 1 to 5 in iteration 6.

8.5.7 Subprogram 8

Cyclomatic complexity: 8, Max loop vars: 10, Max loop arith ops: 0														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 166 initial goals, 4 remaining goals, 6s to simplify, 30s to plan														
2	*rtc*	⊗	⊗											
1	*rinv*	○					●						●	
1	*rtc*	○					●						●	
Iteration 2: 166 initial goals, 4 remaining goals, 6s to simplify, 34s to plan														
2	*rtc*	⊗	⊗											
1	*rinv*	○				●								
1	*rtc*	○				●								
Iteration 3: 166 initial goals, 4 remaining goals, 6s to simplify, 44s to plan														
3	*rtc*	⊗	⊗											
1	*rinv*	⊗			⊗									

In iteration 2 SPADEase requests an engineer to constrain a constant array. In response, appropriate user rules are manually introduced to constrain the array.

In iteration 3 SPADEase fails to make any progress for three run-time check goals. The key step in proving two of these goals is to introduce a rule describing the behaviour of a called function. The third goal becomes provable following the introduction of constraints for a constant array. Nevertheless, neither the Simplifier nor SPADEase proves the goal. The key steps in its proof involve simplifying an implication conclusion and exploiting transitive constraints among hypotheses.

In iteration 3 SPADEase fails to make any progress for a returning invariant goal. The goal is not provable as a variable is under constrained. The program analyser would be able to discover an appropriate constraint. However, the proof failure occurs during

rippling, which does not have critics to trigger specification strengthening. Even with the necessary constraint in place, rippling is unsuccessful. The key step in the proof involves inequality reasoning. SPADEase adopts a form of rippling that is generally more suited to equational conjectures, hindering the development of the proof.

8.5.8 Subprogram 9

Cyclomatic complexity: 7, Max loop vars: 5, Max loop arith ops: 1														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 411 initial goals, 2 remaining goals, 8s to simplify, 142s to plan														
1	*rtc*	○							●					●
1	*rtc*	○	○							●				●
Iteration 2: 421 initial goals, 0 remaining goals, 15s to simplify, 119s to plan														

8.5.9 Subprogram 10

Cyclomatic complexity: 9, Max loop vars: 6, Max loop arith ops: 2														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 411 initial goals, 2 remaining goals, 3s to simplify, 18s to plan														
1	*rinv*	○					●						●	
1	*rtc*	○							●			●		
Iteration 2: 156 initial goals, 1 remaining goals, 3s to simplify, 16s to plan														
1	*rtc*	○						●			●			
Iteration 3: 164 initial goals, 0 remaining goals, 3s to simplify, 15s to plan														

8.5.10 Subprogram 11

Cyclomatic complexity: 8, Max loop vars: 7, Max loop arith ops: 0														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 151 initial goals, 5 remaining goals, 14s to simplify, 43s to plan														
1	*rinv*	○					●						●	
3	*rtc*	○					●						●	
Iteration 2: 178 initial goals, 1 remaining goals, 49s to simplify, 62s to plan														
1	*rtc*	○							●					●
Iteration 3: 180 initial goals, 1 remaining goals, 54s to simplify, 62s to plan														
1	*rtc*	○							●					⊗

In iteration 2, proof failure analysis requests the introduction of tighter constraints for selected variables. Through program analysis, tighter constraints are discovered and introduced through a strengthened invariant. However, in iteration 3, proof failure analysis requests the introduction of tighter constraints for exactly the same variables. The program analyser is unable to deliver tighter constraints, causing SPADEase to make no more progress on this goal.

8.5.11 Subprogram 12

Cyclomatic complexity: 4, Max loop vars: 6, Max loop arith ops: 0															
Goals	**Form**	**Strategy**				**Critic**					**Prog**				
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 57 initial goals, 1 remaining goals, 1s to simplify, 4s to plan														
1	*rinv*	○					●						●	
Iteration 1: 68 initial goals, 1 remaining goals, 1s to simplify, 6s to plan														
1	*rtc*	○						●			●			
Iteration 1: 74 initial goals, 0 remaining goals, 1s to simplify, 5s to plan														

8.5.12 Subprogram 13

Cyclomatic complexity: 3, Max loop vars: 5, Max loop arith ops: 0															
Goals	**Form**	**Strategy**				**Critic**					**Prog**				
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 36 initial goals, 1 remaining goals, 1s to simplify, 3s to plan														
1	*rinv*	○					●						●	
Iteration 2: 40 initial goals, 0 remaining goals, 1s to simplify, 3s to plan														

8.5.13 Subprogram 14

Cyclomatic complexity: 14, Max loop vars: 7, Max loop arith ops: 0															
Goals	**Form**	**Strategy**				**Critic**					**Prog**				
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 265 initial goals, 5 remaining goals, 4s to simplify, 41s to plan														
2	*binv*	○					●						●	
3	*inv*	○					●						●	
Iteration 2: 380 initial goals, 2 remaining goals, 7s to simplify, 84s to plan														
2	*binv*	○					●						○	

In iteration 1 proof failure analysis requests that variables are coupled with their entry variables. Through program analysis, appropriate properties are discovered and introduced through a strengthened invariant. However, due to a known limitation of the SPARK Approach, these invariants are not interpreted correctly. As a consequence, in iteration 2, the strengthened invariants are not present in the remaining goals, 0 seconds. Consequently, proof failure analysis requests that the same variables are coupled with their entry variables. SPADEase is unable to offer further constraints, and the verification fails.

8.5.14 Subprogram 15

Cyclomatic complexity: 8, Max loop vars: 5, Max loop arith ops: 0														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 128 initial goals, 2 remaining goals, 1s to simplify, 11s to plan														
2	*rinv*	●					●						●	
Iteration 2: 144 initial goals, 0 remaining goals, 2s to simplify, 12s to plan														

8.5.15 Subprogram 16

Cyclomatic complexity: 3, Max loop vars: 5, Max loop arith ops: 0														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 48 initial goals, 5 remaining goals, 1s to simplify, 4s to plan														
1	*rinv*	●					●						●	
Iteration 2: 58 initial goals, 5 remaining goals, 1s to simplify, 6s to plan														
1	*rtc*	●						●			●			
Iteration 2: 60 initial goals, 4 remaining goals, 1s to simplify, 4s to plan														

In each iteration, goals are rejected by the targeted_goal method. Two goals relate to proving preconditions while two goals involve real arithmetic.

8.5.16 Subprogram 17

Cyclomatic complexity: 3, Max loop vars: 5, Max loop arith ops: 0														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 37 initial goals, 6 remaining goals, 1s to simplify, 3s to plan														
1	*rinv*	●					●						●	
Iteration 2: 58 initial goals, 5 remaining goals, 1s to simplify, 4s to plan														
1	*rtc*	●						●			●			
Iteration 2: 60 initial goals, 4 remaining goals, 1s to simplify, 3s to plan														

In each iteration, goals are rejected by the targeted_goal method. Three goals relate to proving preconditions while two goals involve real arithmetic.

8.5.17 Subprogram 18

Cyclomatic complexity: 3, Max loop vars: 5, Max loop arith ops: 0														
Goals	**Form**	**Strategy**				**Critic**					**Prog**			
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 39 initial goals, 6 remaining goals, 1s to simplify, 3s to plan														
1	*rinv*	●					●						●	
Iteration 2: 49 initial goals, 6 remaining goals, 1s to simplify, 4s to plan														
1	*rtc*	●						●			●			
Iteration 2: 60 initial goals, 4 remaining goals, 1s to simplify, 3s to plan														

In each iteration, goals are rejected by the targeted_goal method. Three goals relate to proving preconditions while two goals involve real arithmetic.

8.5.18 Subprogram 19

Cyclomatic complexity: 3, Max loop vars: 4, Max loop arith ops: 1															
Goals	**Form**	**Strategy**				**Critic**					**Prog**				
		ef	**rc**	**tr**	**ri**	**cc**	**ce**	**cv**	**tc**	**fc**	**ty**	**lr**	**ts**	**ic**
Iteration 1: 34 initial goals, 1 remaining goals, 1s to simplify, 6s to plan															
1	*rtc*	○							●					●	
Iteration 1: 38 initial goals, 0 remaining goals, 1s to simplify, 2s to plan															

8.6 Overall Analysis

An overall analysis of the evaluation is performed. The relationship between subprogram complexity and proof effort is considered in §8.6.1. The relationship between subprogram complexity and the number of verification iterations is considered in §8.6.2.

8.6.1 Comparing Complexity and Proof Effort

SPADEase is an enhancement of the SPARK Approach, typically operating over a number of verification iterations. The effort involved in proving that a subprogram is free from exceptions is calculated as the total simplification and planning time across all iterations. Note that, since the program analysis heuristics were manually simulated for the industrial subprograms, their execution times are not available and so are not considered.

There is no definitive method for measuring subprogram complexity. Thus, a complexity measure was incrementally developed as discussed below.

- **First complexity measure** - Cyclomatic complexity reports on the path complexity of a subprogram. This metric is used in calculating the first complexity measure as follows:

$$\textit{first complexity} = \textit{cyclomatic complexity} \tag{8.1}$$

  The first complexity measure is compared against proof effort in Figure 8.8. The spread of results suggest that additional factors are influencing proof effort.

- **Second complexity measure** - The maximum number of variables inside a loop is determined. The metric is used in calculating the second complexity measure as follows:

$$\textit{second complexity} = \textit{cyclomatic complexity} * (\textit{max loop vars} + 1) \tag{8.2}$$

The second complexity measure is compared against proof effort in Figure 8.9. Through considering the role of variables, there is a closer relationship between complexity and proof effort. However, it still appears that additional factors are influencing proof effort.

- **Third complexity measure** - The maximum number of distinct arithmetic operators inside a loop is determined. The metric is used in calculating the third complexity measure as follows:

$$\begin{aligned} \textit{third complexity} = \textit{cyclomatic complexity} * (\textit{max loop vars} + 1) * \\ (\textit{max loop arith ops} + 1) \end{aligned} \tag{8.3}$$

The third complexity measure is compared against proof effort in Figure 8.10. Through considering the role of arithmetic operators, the relationship between complexity and proof effort is clearer. Two particularly outlying values, corresponding to *Subprogram 3* and *Subprogram 10*, have both high complexity and low proof effort. In both cases, the Simplifier is particularly efficient and the proof plans quickly identify the need for invariant discovery. It is speculated that a richer consideration of term complexity would deliver a stronger relationship to overall proof effort.

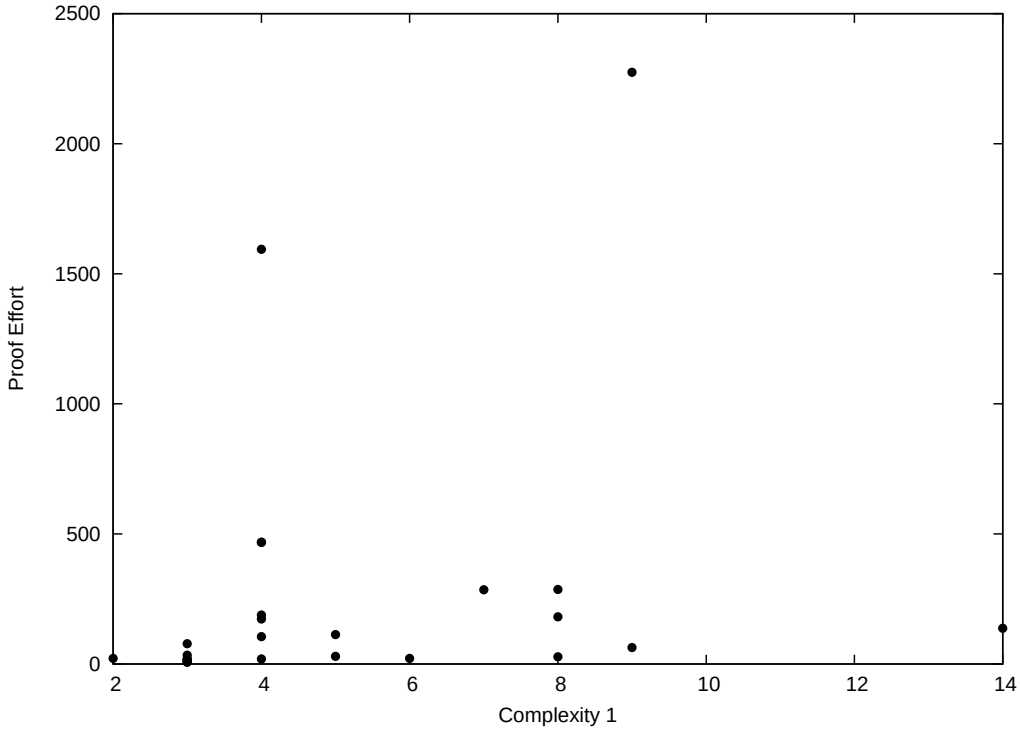


Figure 8.8: First complexity measure against proof effort

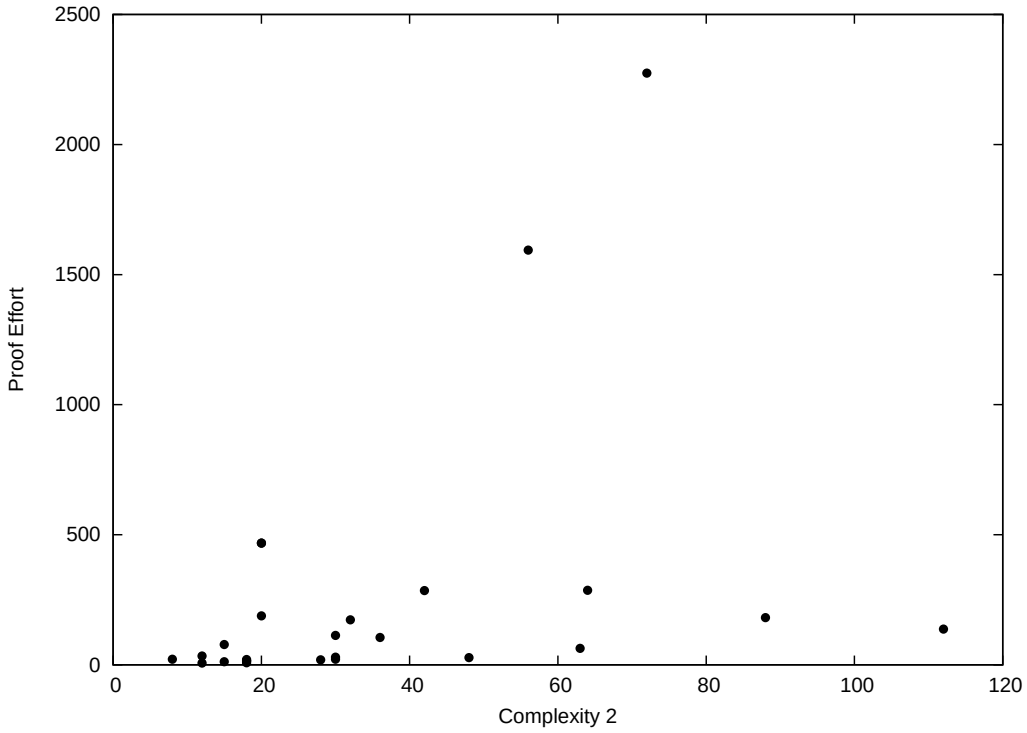


Figure 8.9: Second complexity measure against proof effort

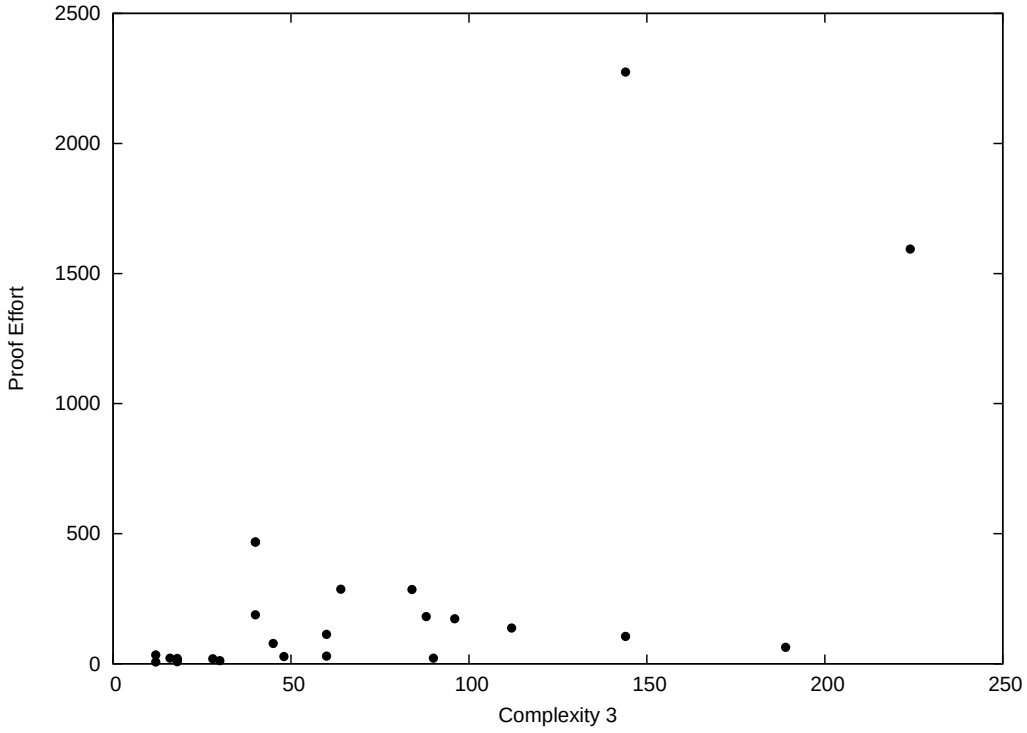


Figure 8.10: Third complexity measure against proof effort

8.6.2 Comparing Complexity and Iterations

The third complexity measure is compared against the number of verification iterations Figure 8.11. As complexity falls, there is a trend for fewer iterations.

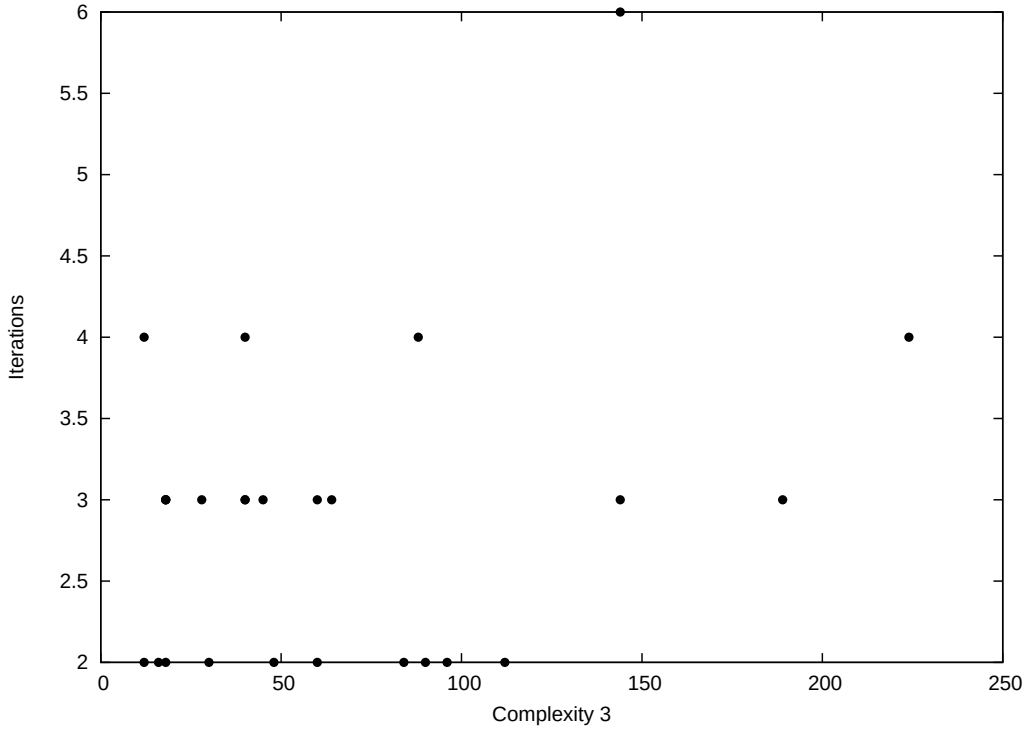


Figure 8.11: Third complexity measure against iterations

Chapter 9

Conclusions

In this concluding chapter the specific contributions and wider implications of the thesis are considered. The main contributions of our approach are considered in §9.1, highlighting related work. In §9.2 limitations and future work are considered. Finally, a closing summary is made in §9.3.

9.1 Contributions

In §1.2 the six main contributions of this thesis were outlined. Below, these contributions are repeated, highlighting how each has been achieved and describing related work.

Configurable and Sound

> *Present a configurable and justifiably sound approach to software verification.*

Proof planning [Bun88] makes a clear distinction between its supporting infrastructure and its controlling heuristics. Our approach maintains this distinction for the task of program verification. Proof discovery techniques are captured as proof plans while invariant discovery techniques are captured as program analysis heuristics. Consequently, the approach may be readily configured by modifying its heuristics accordingly.

Where verification tools may be subject to configuration, the soundness of these configurations becomes a key concern. Proof planning addresses this concern by making a clear separation between plausible and demonstrative reasoning. Our approach maintains this distinction by strictly operating within a sound program verification environment. All discovered proofs are checked inside a proof checker, and all discovered invariants are checked indirectly through program verification. The selective positioning of soundness concerns is also seen in program generation [WH99a]. It is typically impractical to verify the correctness of a program generation system. Instead, translation validation is performed, verifying the correctness of each generated program. Thus, the approach positions soundness concerns in a more tractable location.

Cooperative Integration

Demonstrate that more targeted and effective automation can be achieved through the cooperative integration of distinct technologies.

Program verification involves both proof discovery and invariant discovery. Our approach tackles these related tasks through a cooperative integration. Where proof discovery is unsuccessful, proof failure analysis attempts to determine the cause of the failure. Where the failure is traced to a weakness in the specification, targeted invariant discovery is triggered. The collaborative style means that our approach focuses on addressing the genuine verification challenge. A related approach is employed by Houdini [FL01] to support invariant discovery in ESC/Java [FLL+02]. Houdini initially discovers candidate invariants without considering the verification problem. As a consequence, the candidate invariants may not be relevant. To address this concern, the candidate invariants are introduced and program verification is performed. Those invariants that are incorrect or irrelevant are removed. Thus, Houdini exploits the genuine verification challenge to filter its discovered invariants. Nevertheless, the collaboration is limited and retrospective. Our approach works towards the verification challenge, addressing specific weaknesses in the specification through targeted invariant discovery. Consequently, our approach naturally delivers pertinent invariants, without requiring a filtering phase.

Proof Discovery

Present proof plans that support the verification of exception freedom.

Proof plans are presented that support the automated verification of exception freedom. Separate plans are developed for VCs corresponding to run-time checks and invariants. The proof plans are expressed at a high-level in an explicit method-language, facilitating their understanding and reuse. The feasibility of this reuse is exploited in our proof plans. Rippling [BBHI05] was originally developed to support proof by induction. It is reused in our approach, to prove loop invariant VCs.

Invariant Discovery

Present invariant discovery heuristics that support the verification of exception freedom.

Program analysis heuristics are presented that automatically discover invariant properties suitable for advancing a proof of exception freedom. The heuristics are expressed in a consistent form as program analysis methods, facilitating their understanding and reuse. A key technique employed is the generation and solving of recurrence relations to generate invariant properties. The Runcheck system [Ger81] adopted a similar approach. Transformations made to program variable were collected as *change vectors*, essentially

the same as recurrence relations. Our approach extends the technique by considering program context, nested loops, and generating bounded constraints. Further, our approach exploits a powerful recurrence relation solver.

Implementation as SPADEase

Implement our approach as SPADEase.

Our approach has been implemented as an extension to the SPARK Approach as SPADEase. The primary components of SPADEase are a proof planner and a program analyser. These systems have been developed to directly support our approach. Where appropriate, SPADEase has reused existing components. These include the Clam method-language [BvHHS90], the clp(FD) constraint solver [COC97], the YACAS computer algebra system [YAC], a SPARK grammar, a SPARK tokeniser, the Stratego program transformation system [Vis01], and the PURRS recurrence relation solver [PUR].

Evaluation

Evaluate SPADEase against both textbook and industrial subprograms.

Our approach is evaluated against industrial subprograms. As our program analyser only supports a subset of SPARK, the program analysis aspect is simulated. Of the 20 subprograms considered, our approach completes the verification of exception freedom for 12. In 3 cases, our approach completes the verification of all considered exception freedom goals, however there are residual goals related to real arithmetic or verifying preconditions which are not considered by our approach. In 1 case, verification was not possible due to a limitation of the SPARK Approach. In the remaining 3 cases, our approach advances the verification of exception freedom, but not to completion.

9.2 Limitations and Future Work

Limitations of our approach are considered in the sections below, indicating how these may be addressed through future work.

9.2.1 Support Preconditions and Postconditions

Preconditions and postconditions may provide valuable constraints in analysing a subprogram. However, SPADEase currently ignores these. The limitation may be addressed through the introduction of program analysis heuristics that exploit these constraints. Similarly, SPADEase only considers specification strengthening through invariant discovery. Both precondition and postcondition discovery may be appropriate to advance a verification effort. The limitation might be addressed through the adoption of more sophisticated

abstract predicate satisfies. As modifications to preconditions and postconditions change the specification, each suggested change would need to be checked by an engineer.

9.2.2 Adopt Tactic Based Theorem Prover

SPADEase delivers both configurability and soundness by relying on the soundness of the SPARK Approach. However, as detailed in §B.4, significant modifications were made to the Checker. The changes were required to achieve an effective integration, as the Checker is not suited to automated control. Such modifications have the potential to undermine soundness and thus would require rigorous verification and validation. The need for modifications and certification may be avoided by selecting a tactic based theorem prover as a proof checker. Such a prover would be suited to automated control without any modifications.

9.2.3 Automated Lemma Discovery

As described in §B.4.4, a small collection of theorems were introduced to support the application of SPADEase. In general, the discovery and proof of such intermediate lemmas can hinder a verification effort. This limitation may be addressed by extending SPADEase to support automated lemma discovery. Proof failure analysis might identify the structure of a missing lemma, triggering targeted lemma discovery. In particular, the automated discovery of lemmas to advance rippling has previously been investigated [IB96].

9.3 Summary

Our approach provides an effective environment for automated program verification. The approach strictly enhances an existing program verification environment. By doing so, the approach is simultaneously configurable and sound. Our approach addresses the entire verification challenge, including both proof discovery and invariant discovery. Proof discovery is achieved via a proof planner and invariant discovery is achieved via a program analyser. Significantly, the two components are cooperatively integrated such that they work together in addressing genuine verification problems. The approach is tailored to automate the verification of exception freedom in the SPARK Approach. The approach is realised as SPADEase and has been favourably evaluated against both industrial and textbook subprograms.

Appendix A

PolishFlag Interactive Proof

A.1 Introduction

The partial correctness of the PolishFlag subprogram is considered in §4.4.5. The Simplifier is able to prove a few conclusions, with the remaining VCs being stored in an SIV file, as shown in Figure 4.13. The remaining VCs have been interactively proved inside the Checker. The corresponding CMD file is split across Figure A.1 and Figure A.2.

```
1.
consult 'permutation.rul'.
consult 'polishflag.rls'.
unwrap c#1.
prove c#1 by implication.
infer false using inference(2).
prove c#1 by contradiction.
done.
unwrap c#2.
prove c#1 by implication.
infer false using inference(2).
prove c#1 by contradiction.
done.
infer c#7 using permutation(1).
2.
unwrap c#1.
prove c#1 by implication.
unwrap h#1.
inst int_q__1.
prove c#1 by cases on int_q__1=i or int_q__1<i.
replace c#1: int_q__1 by i using eq(1).
yes.
no.
done.
infer int_q__1 <= i - 1 using inequals(74).
yes.
yes.
standardise c#1.
yes.
done.
forwardchain h#10.
done.
3.
unwrap c#1.
prove c#1 by implication.
infer int_q__1 < i using transitivity(19).
infer i <= j - 1 using inequals(74).
```

Figure A.1: PolishFlag subprogram interactive proof (CMD) [1 of 2]

```
yes.
yes.
stand c#1.
yes.
done.
infer int_q__1 < i using transitivity(19).
infer int_q__1 < j - 1 using inequals(31).
infer int_q__1 <> j - 1 using inequals(33).
replace c#1: element(update(update(flag, [i], element(flag, [j - 1])),
             [j - 1], element(flag, [i])), [int_q__1]) by
             element(update(flag, [i], element(flag, [j - 1])),
             [int_q__1]) using array(3).
yes.
no.
infer int_q__1 <> i using transitivity(30).
replace c#1: element(update(flag, [i], element(flag, [j - 1])),
             [int_q__1]) by element(flag, [int_q__1]) using array(3).
yes.
no.
unwrap h#1.
inst int_q__1.
forwardchain h#16.
done.
unwrap c#2.
prove c#1 by implication.
infer (element(flag, [i]) = red) using enum(16).
infer j-1 <= int_r__1 using inequals(15).
prove c#1 by cases on j-1 = int_r__1 or j-1 < int_r__1.
done.
infer j<= int_r__1 using inequals(102).
infer i < int_r__1 using transitivity(20).
infer i <> int_r__1 using transitivity(30).
replace c#1: element(update(flag, [i], element(flag, [j - 1])),
             [int_r__1]) by element(flag, [int_r__1]) using array(3).
yes.
no.
unwrap h#2.
inst int_r__1.
forwardchain h#18.
done.
replace c#7: permutation(update(update(flag, [i], element(flag, [j - 1])),
             [j - 1], element(flag, [i])), flag__OLD) by
             permutation(update(update(flag, [j - 1], element(flag, [j - 1])),
             [i], element(flag, [i])), flag__OLD) using permutation(3).
yes.
no.
replace c#7: update(flag, [j-1], element(flag, [j - 1])) by flag using array(2).
yes.
no.
replace c#7: update(flag, [i], element(flag, [i])) by flag using array(2).
yes.
no.
done.
4.
unwrap c#1.
inst i.
done.
unwrap c#4.
unwrap h#2.
inst int_r__1.
prove c#1 by implication.
infer j <=  int_r__1 using transitivity(1).
forw h#9.
done.
exit.
```

Figure A.2: PolishFlag subprogram interactive proof (CMD) [2 of 2]

Appendix B

Modifying the SPARK Toolset

B.1 Introduction

This chapter describes modifications made to the SPARK toolset to support integration with SPADEase.

B.2 Modifications to the Examiner

The Examiner describes quantified expressions in VCs through the following expressions:

```
for_all (Variable : Type, Expression)
for_some (Variable : Type, Expression)
```

The white space between the quantifier and its arguments creates complications in parsing these expressions. Thus, the Examiner was modified to prevent the generation of this white space.

B.3 Modifications to the Simplifier

The Simplifier receives the initial VCs, performs simplification, and generates corresponding remaining VCs. In generating the remaining VCs, the Simplifier renumbers the conclusions of each VC. This feature makes it difficult to automatically associate simplified conclusions in the remaining VCs with their original form in the initial VCs. To resolve this, the Simplifier was modified to include an additional switch `/norenum`. The switch suppresses the renumbering of conclusions when generating the remaining VCs.

B.4 Modifications to the Checker

The Checker is used by SPADEase to check the correctness of discovered proof plans. To make this process both feasible and practical, various modifications are made to the Checker as detailed below.

B.4.1 Principled Proof Checking Interface

The Checker operates as an automated proof checker by receiving proof commands from a file rather than interactively from an engineer. The success or otherwise of the proof effort is stored as part of the Checker proof log. To integrate more effectively with SPADEase a more principled proof checking interface was introduced. The Checker was modified to include an additional switch `/tame` as described below:

- `/tame` *VCGFile VCId ConcId ProofCommandFile ResultFile*
  - *VCGFile* - Targeted VCG file.
  - *VCId* - Targeted VC.
  - *ConcId* - Targeted conclusion.
  - *ProofCommandFile* - Proof command file to be checked.
  - *ResultFile* - Location to store the result of the proof checking. The provided proof command file is executed to try and discharge the targeted conclusion of the targeted VC of the targeted VCG file. Where successful, the result file will contain '`true.`', otherwise '`false.`'.

B.4.2 Improve Predictability

During a proof session the Checker is proactive, automatically proving conclusions that are within its capabilities. Consequently, it is difficult to predict exactly how the Checker will behave following a proof command. For SPADEase to correctly translate discovered proof plans into Checker proof command files it is essential that the Checker behaves in a predictable manner. To achieve this, where the `/tame` switch is supplied, the Checker was modified to deactivate all proactive proof automation.

B.4.3 Richer Proof Commands

Typically, a proof planner is coupled to a tactic based theorem prover, as tactics provide a powerful mechanism for executing a discovered proof plan. However, the Checker is not a tactic based theorem prover. This mismatch created complications in the technical task of translating discovered proof plans into Checker proof command files. In particular, a single intuitive proof step might be translated as an involved nesting of multiple proof commands. Consequently, the resulting proof command files are both difficult to generate and to comprehend. To avoid these complications, the Checker was extended to support additional proof commands where the `/tame` switch is present. The following minor proof commands were added:

- `tame_subgoal_on_exp` *BoolExp* - Generate a subgoal containing the hypotheses of the current goal and the single conclusion as the Boolean expression *BoolExp*.

- `tame_subgoal_on_conc` *ConcId* - Generate a subgoal containing the hypotheses of the current goal and the single conclusion associated with the identifier *ConcId*.
- `tame_done` - Appeal to the automated capabilities of the Checker to prove any conclusion in the current goal.
- `tame_all_done` - The current goal is closed if it does not contain any conclusions.
- `tame_finish` - Exit the Checker, and store the success of the proof in the result file.

Further, a powerful rewrite command, with four alternative modes of operation, was introduced:

- `tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `with` *LHSExp* `to` *RHSExp* `if` *Condition* `using` *RewriteRuleId* `in` *Direction* -
  Rewrite using a previously loaded rewrite rule.
- `tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `with` *LHSExp* `to` *RHSExp* `if` *Condition* `from` *HypExp* `in` *Direction* -
  Rewrite using a hypothesis as a rewrite rule.
- `tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `where` *HypExp* -
  Rewrite using a hypothesis as an alternative expression for true.
- `tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `with` *EvalExp* `is` *Value* -
  Rewrite an expression to its evaluated value.

Where the meaning of each argument is as detailed below:

- *HypOrConc* - denotes whether a hypothesis or conclusion is being rewritten as *hyp* or *conc* respectively.
- *WholeExp* - The whole expression of the selected hypothesis or conclusion.
- *Pos* - The position of the subexpression within the whole expression that is to be rewritten. The position is expressed as a list of integers, describing a path through the expression structure.
- *LHSExp* - The expression being rewritten from. This must match the subexpression at position *Pos* of *WholeExp*.
- *RHSExp* - The expression being rewritten to.
- *Condition* - A condition associated with a rewrite rule. For the rule to be applied, the condition must be either *true* or match with a hypothesis.
- *RewriteRuleId* - The unique identifier of a rule. The corresponding rule, adjusted for *Direction*, must match with *LHSExp*, *RHSExp* and *Condition*.

- *Direction* - Is either *normal* or *reversed* to denote the direction that a rewrite rule is to be applied.
- *HypExp* - The expression of a selected hypothesis.
- *EvalExp* - The evaluatable expression being rewritten from. This must match the subexpression at position *Pos* of *WholeExp*.
- *Value* - The result of evaluating an expression. This must match the result of evaluating *EvalExp*.

The rewrite command may be applied to hypotheses or conclusions. To minimise polarity concerns, only the whole expression or a top level conjunct may be rewritten. Significantly, this restriction excludes the rewriting of quantified expressions.

B.4.4 Adding Theorems Through User Rules

As discussed in §6.6, properties and definitions are held in external rule files. A large number of standard rules are available. These rules are extracted as theorems and used to support the vast majority of our proof plans. However, a few desired theorems are not available as part of the standard rules. To resolve this, appropriate theorems are introduced through user rules, as detailed in the sections below.

Alternative Views

The select_alt_view_rule predicate draws upon the available rewrite rules to explore alternative views of an expression. To support this predicate, theorems are introduced that describe the preservation of an expression and the rotation of inequality expressions:

$$\forall(y : \_. (true \rightarrow (y = y))) \tag{B.1}$$

$$\forall(y, z : integer. (true \rightarrow ((y > z) = (z < y)))) \tag{B.2}$$

$$\forall(y, z : integer. (true \rightarrow ((y \geq z) = (z \leq y)))) \tag{B.3}$$

Decompose Conjuncts

In the Checker, conjuncts are decomposed through simplification strategies. Such strategies are unpredictable and thus not appropriate for proof checking. Instead, specific theorems are introduced to support the decomposition of conjuncts:

$$\forall(y, z : boolean. (true \rightarrow ((y \wedge z) \rightarrow y))) \tag{B.4}$$

$$\forall(y, z : boolean. (true \rightarrow ((y \wedge z) \rightarrow z))) \tag{B.5}$$

Transitivity Decomposition

In progressing a transitivity step, rewrite rules are required that decompose inequality expressions. To support this process, the following theorems are introduced:

$$\forall(x, y, z : integer.\ (true \rightarrow ((x \geq y) \wedge (0 \geq z)) \rightarrow (x \geq (y + z)))) \tag{B.6}$$

$$\forall(x, y, z : integer.\ (true \rightarrow ((x \leq y) \wedge (0 \leq z)) \rightarrow (x \leq (y + z)))) \tag{B.7}$$

Transitivity Unblock

In unblocking a transitivity step, rewrite rules are required that ground partially instantiated inequality expressions. To support this, the following theorems are introduced:

$$\forall(x : integer.\ (true \rightarrow (true \rightarrow (x \geq x)))) \tag{B.8}$$

$$\forall(x : integer.\ (true \rightarrow (true \rightarrow (x \leq x)))) \tag{B.9}$$

Rippling

For rippling, structurally preserving rewrite rules are required. However, some key standard rules are not expressed in a structurally preserving form. To resolve this, the following theorems are introduced:

$$\forall(x, y, z : integer.\ (true \rightarrow (((x + y) + z) = ((x + z) + y)))) \tag{B.10}$$

$$\forall(x, y, z : integer.\ (true \rightarrow (((x - y) + z) = (x + z) - y))) \tag{B.11}$$

$$\forall(x, y, z : integer.\ (true \rightarrow (((x + y) * z) = ((x * z) + (y * z))))) \tag{B.12}$$

Disjunctive Normal Form

The disj_norm_form method draws upon the following theorems to transform an expression into disjunctive normal form:

$$\forall(y, z : boolean.\ (true \rightarrow ((y \rightarrow z) = ((\neg y \vee z))))) \tag{B.13}$$

$$\forall(y, z : boolean.\ (true \rightarrow ((y \leftrightarrow z) = ((y \rightarrow z) \wedge (z \rightarrow y))))) \tag{B.14}$$

$$\forall(z : boolean.\ (true \rightarrow ((\neg(\neg(z))) = z))) \tag{B.15}$$

$$\forall(y, z : boolean.\ (true \rightarrow ((\neg(y \wedge z)) = ((\neg(y) \vee \neg(z)))))) \tag{B.16}$$

$$\forall(y, z : boolean.\ (true \rightarrow ((\neg(y \vee z)) = (\neg(y) \wedge \neg(z))))) \tag{B.17}$$

$$\forall(x, y, z : boolean.\ (true \rightarrow (((x \vee y) \wedge z) = ((x \wedge z) \vee (y \wedge z))))) \tag{B.18}$$

$$\forall(x, y, z : boolean.\ (true \rightarrow ((x \wedge (y \vee z)) = ((x \wedge y) \vee (x \wedge z))))) \tag{B.19}$$

Appendix C

Method-Language

C.1 Introduction

The method-language supports the expression of proof plans. The method-language is composed from a number of *predicates*. The general form of these predicates is shown in §C.2, while the predicates themselves are detailed in the remainder of this appendix.

C.2 Method-Language Predicates

Each method-language predicate takes the following general form:

predicate($Mode_1\ Arg_1, \ldots, Mode_n\ Arg_n$) *Provenance*

Where predicate is the name of the predicate, Arg_i are its arguments and $Mode_n$ describes the mode of each argument as below:

Mode	Description
+	An input value.
−	An output value.
?	Either an input value or an output value.

Provenance identifies those predicates that have been reused from elsewhere. The symbol Ⓟ indicates that the predicate originated from Prolog, while the symbol © indicates that the predicate originated from Clam.

Method-language predicates occur in a list of the general form:

$$[Predicate_1, \ldots, Predicate_z] \tag{C.1}$$

Each predicate may have multiple solutions, which are explored through *backtracking*. To begin, the predicates are considered in order, from 1 to z, accepting the first solutions found. Then, backtracking is performed, considering the predicates in reverse order, from z to 1, seeking the first predicate y that has an alternative solution. Where found, the alternative solution is explored by reconsidering its subsequent predicates in order, from $y + 1$ to z. Backtracking is repeated until all alternative solutions have been explored.

C.3 Composition

C.3.1 cut

cut Ⓟ

By default, every successful predicate is explored through backtracking. This predicate instructs the caller to dismiss all alternative solutions that exist prior to the cut. The predicate is valuable where many potential solutions are possible, but only a single solution is sought.

C.3.2 not

not(+*Predicate*) Ⓟ

The predicate is successful where *Predicate* is unsuccessful.

C.4 Proof Planning

C.4.1 abort_plan

abort_plan(+*FailureCritique*)

This predicate aborts the current plan, reporting failure critique *FailureCritique.*

C.4.2 plan_lemmas

plan_lemmas(+*HypList*, +*LemmaList*, +*Strategy*, +*ProvedList*, −*Tactic*)

This predicate plans each lemma in *LemmaList* as a separate goal[1]. Each lemma forms the conclusion of the goal, while its hypotheses are *HypList*. The initial strategy for planning each lemma is *Strategy*. The proved lemmas are returned as *ProvedList* alongside a supporting tactic *Tactic* that introduces the lemmas as hypotheses at the object-level.

C.4.3 write_line

write_line(+*Text*) Ⓟ

This predicate sends *Text* to the standard output.

[1]The predicate involves nested proof planning during method application, which is not supported by our proof planner. Instead, nested planning is simulated through two communicating methods. The first method queues the lemmas to be planned, while the second method collects the results. While inelegant, the technique retains the essential behaviour of this predicate.

C.5 List Processing

C.5.1 append

append(?*FirstList*, ?*SecondList*, ?*CombinedList*) Ⓟ

This predicate is successful were appending *SecondList* to the end of *FirstList* generates *CombinedList*. The predicate may be used to append two lists together or to backtrack over every pair of lists that, when appended, produce a particular list.

C.5.2 filter_duplicates

filter_duplicates(+*ItemList*, −*DuplItemList*, −*UniqItemList*)

This predicate filters duplicate items. Every item that appears more than once in *ItemList* is reported once in *DuplItemList*. Every item that appears in *ItemList* is reported once in *UniqItemList*.

C.5.3 select

select(?*Item*, ?*ItemList*, ?*RemItemList*) Ⓟ

This predicate is successful where *Item* is in *ItemList*, and the remaining items are *RemItemList*. The predicate may be used to check that an item is in a list, to remove an item from a list or to backtrack over all of the items in a list.

C.6 Plan Features

C.6.1 add_under_constrained_vars

add_under_constrained_vars(+*SubprogramName*, +*UnderConstrainedVarList*)

This predicate extends the global contextual information associated with a plan. It adds the under constrained variables *UnderConstrainedVarList* alongside the name of the subprogram corresponding to the goal as *SubprogramName*.

C.6.2 get_goal_category

get_goal_category(−*GoalCat*)

This predicate reports the category of the goal as *GoalCat*. As detailed in §6.5.3, the global contextual information associated with the plan includes traceability information that describes how the goal relates to the source code. The goal categories are derived from the traceability information as shown in Figure C.1 and described below:

- *rtc* - A transition from any cut-point to a run-time check.
- *rinv* - A transition from an invariant returning to the same invariant.

- *binv* - A transition between different invariants.
- *other* - All other transitions.

```
<GoalCat> ::=rtc {where <Trace> =
                      betweenPath(_, check(runtime, _))} |
              rinv {where <Trace> =
                      betweenPath(assertion(_, LineInt), assertion(_, LineInt))} |
              binv {where <Trace> =
                      betweenPath(assertion(_, FromLineInt), assertion(_, ToLineInt)) ∧
                      (FromLineInt ≠ ToLineInt)} |
              other
```

Figure C.1: Goal Categories

C.6.3 match_global_context

match_global_context(?*MatchExp*)

This predicate is successful where the expression *MatchExp* matches an item in the global contextual information associated with the plan.

C.7 Goal Features

C.7.1 add_constraining_vars

add_constraining_vars(+*LocalContextList*, +*VarList*, −*ExtendedLocalContextList*)

This predicate extends the local context information *LocalContextList* to record the constraining variables *VarList* as *ExtendedLocalContextList*.

C.7.2 get_constraining_vars

get_constraining_vars(+*LocalContextList*, −*VarList*)

This predicate reports all constraining variables recorded in the local context information *LocalContextList* as *VarList*.

C.8 Goal Patterns

C.8.1 aux_vars

aux_vars(+*Exp*, −*AuxlVarList*, −*TotalInt*)

As detailed in §6.5.1, auxiliary variables are introduced to support program analysis. This predicate searches expression *Exp* for auxiliary variables, returning those found as *AuxlVarList* alongside the total number found as *TotalInt*.

C.8.2 unconstrained_consts_vars

unconstrained_consts_vars(+*Goal*, −*UnconstrainedConstList*, −*UnconstrainedVarList*)

This predicate inspects *Goal* to identify every integer constant and variable that lacks an explicit upper or lower bound as *UnconstrainedConstList* and *UnconstrainedVarList* respectively. Note that an exception is made for entry variables, as these are often constrained indirectly, by being related to their corresponding standard variable.

C.8.3 uncoupled_entry_vars

uncoupled_entry_vars(+*Goal*, −*UncoupledVarList*)

This predicate inspects *Goal* to identify all uncoupled variables as *UncoupledVarList*. Such variables have a corresponding entry variable yet there is no hypothesis describing the relationship between the two variables. For further details on entry variables, see the viable_goal method in §E.13.

C.8.4 under_constrained_vars

under_constrained_vars(+*Goal*, −*UnderConstrainedVarList*)

This predicate inspects *Goal* via a constraint solver, identifying under constrained variables as *UnderConstrainedVarList.* For further details on the integration of a constraint solver see the viable_goal method in §E.13.

C.9 Analyse Expressions

C.9.1 binary_explode

binary_explode(+*Exp*, −*Op*, −*LeftExp*, −*RightExp*)

This predicate is successful where expression *Exp* is a binary operation. In this case, the predicate returns the binary operator as *Op,* and the left and right expressions as *LeftExp* and *RightExp* respectively.

C.9.2 conjunct_at

conjunct_at(+*Exp*, −*Pos*, −*ConjExp*)

This predicate returns every distinct conjunct within *Exp* as *ConjExp* alongside its position *Pos*. For example, given the expression $X \wedge (Y \wedge Z)$, the conjuncts returned will be *X*, *Y* and *Z*.

C.9.3 elim_bounded_var

elim_bounded_var(+*HypList*, +*Var*, +*Exp*, −*NewExp*)

This predicate eliminates an occurrence of *Var* in *Exp* returning the new expression as *NewExp*. The elimination is supported through interval reasoning. The hypotheses *HypList* are explored to identify a unique upper and lower bound for *Var*. Then, depending on the structure of *Exp, Var* in *Exp* is replaced with these bounds, generating *NewExp*.

C.9.4 eval_exp

eval_exp(+*EvalExp*, −*Value*)

This predicate is successful where the expression *EvalExp* can be evaluated, reporting the result of its evaluation as *Value*. Both integer and Boolean evaluation is considered, as shown in Figure C.2.

<*EvalExp*> ::= <*BoolEval*>
<*BoolEval*> ::= <*BoolEval*> $\wedge$ <*BoolEval*> | <*BoolEval*> $\vee$ <*BoolEval*> |
<*BoolEval*> $\rightarrow$ <*BoolEval*> | <*BoolEval*> $\leftrightarrow$ <*BoolEval*> |
$\neg$<*BoolEval*> | <*EqEval*> | <*boolean*>
<*EqEval*> ::= <*ExpEval*> = <*ExpEval*> | <*ExpEval*> $\neq$ <*ExpEval*> |
<*ExpEval*> < <*ExpEval*> | <*ExpEval*> $\leq$ <*ExpEval*> |
<*ExpEval*> > <*ExpEval*> | <*ExpEval*> $\geq$ <*ExpEval*>
<*ExpEval*> ::= <*ExpEval*> $*$ <*ExpEval*> | <*ExpEval*> $**$ <*ExpEval*> |
<*ExpEval*> *div* <*ExpEval*> | <*ExpEval*> *mod* <*ExpEval*> |
<*ExpEval*> + <*ExpEval*> | <*ExpEval*> − <*ExpEval*> |
−<*ExpEval*> | <*integer*>

Figure C.2: Evaluatable expressions

C.9.5 exp_at

exp_at(+*Exp*, −*Pos*, −*SubExp*) ©

This predicate returns every subexpression in expression *Exp* as *SubExp* alongside its position *Pos*.

C.9.6 exp_explode

exp_explode(+*Exp*, +*Op*, −*ExpList*)

This predicate recursively splits expression *Exp* between occurrences of *Op*. The resulting expressions are returned as *ExpList*.

C.9.7 find_replace

find_replace(+*Exp*, +*FindExp*, +*ReplaceExp*, −*ModifiedExp*)

This predicate replaces every occurrence of *FindExp* in *Exp* with *ReplaceExp* generating the modified expression *ModifiedExp*.

C.9.8 ground

ground(+*Exp*) Ⓟ

This predicate succeeds where expression *Exp* does not contain any meta-variables.

C.9.9 int_bound_var

int_bound_var(+*HypList*, +*Exp*, −*LowerInt*, −*UpperInt*)

This predicate searches the hypotheses *HypList* to discover tight lower and upper numeric bounds for the integer expression *Exp* as *LowerInt* and *UpperInt* respectively. The predicate is only successful where both an upper and lower bound can be found. The mechanism for finding numeric bounds is described in Figure C.3.

To find the lower bound, each hypothesis is matched as below, to discover a collection of candidate bounds. The largest candidate bound is selected as the tightest lower bound.

Hypothesis match	**Candidate bound**
ExpInt ≥ *BoundInt*	*BoundInt*
BoundInt ≤ *ExpInt*	*BoundInt*
ExpInt > *BoundInt*	*BoundInt* + 1
BoundInt < *ExpInt*	*BoundInt* + 1

To find the upper bound, each hypothesis is matched as below, to discover a collection of candidate bounds. The smallest candidate bound is selected as the tightest upper bound.

Hypothesis match	**Candidate bound**
ExpInt ≤ *BoundInt*	*BoundInt*
BoundInt ≥ *ExpInt*	*BoundInt*
ExpInt < *BoundInt*	*BoundInt* − 1
BoundInt > *ExpInt*	*BoundInt* − 1

Figure C.3: Discover tight numeric bounds

C.9.10 is_inequality_op

is_inequality_op(+*Op*)

This predicate is successful where operator *Op* is the inequality $<$, $\leq$, $\geq$ or $>$.

C.9.11 is_int

is_int(+*Exp*) Ⓟ

This predicate is successful if expression *Exp* is an integer.

C.9.12 prog_var_exps

prog_var_exps(+*Exp*, −*ProgVarExpList*, −*TotalInt*)

This predicate searches expression *Exp* for program variable expressions. These expressions are returned as *ProgVarExpList* alongside the total number found as *TotalInt*. The program variable expressions considered are listed below:

- **Variable** - *VarRef*
- **Element of an array** - *element*(*ArrayRef*, *IndexList*)
- **Field of a record** - *fld__FieldRef*(*RecordRef*)

C.9.13 remove_real_exps

remove_real_exps(+*InExpList*, −*OutDiscreteExpList*)

This predicate removes those expressions in *InExpList* that involve real arithmetic, returning the remaining expressions as *OutDiscreteExpList*.

C.9.14 replace_at

replace_at(+*Exp*, +*Pos*, +*ReplaceExp*, −*ModifiedExp*) Ⓒ

This predicate replaces the subexpression at position *Pos* of *Exp* with *ReplaceExp* generating the modified expression *ModifiedExp*.

C.9.15 simple_linear_exp_var

simple_linear_exp_var(+*Exp*, −*ProgVarExp*)

This predicate is successful where *Exp* is a simple linear expression, as described in Figure C.4. Where successful, the single program variable expression is returned as *ProgVarExp*.

<*SimpleLinearExp*> ::= <*MultPart*>
<*MultPart*> ::= <*TerminalPart*> ∗ <*SumPart*> |
<*Sum*> ∗ <*TerminalPart*> |
<*SumPart*>
<*SumPart*> ::= <*TerminalPart*> + <*TerminalPart*> |
<*TerminalPart*> − <*TerminalPart*> |
<*TerminalPart*>
<*TerminalPart*> ::= <*integer*> |
<*program variable expression*> {*must* occur *once*.
Expression defined by prog_var_exps.}

Figure C.4: Simple linear expression

C.9.16 solve_for_var

solve_for_var(+*Eq*, +*Var*, −*SolvedEq*)

This predicate is successful where the equality *Eq* can be transformed into an equality for *Var*, returning the resulting equality as *SolvedEq*. The predicate is supported through a computer algebra system. For further details on the integration of a computer algebra system see the solve_eq_hyp_for_var method in §E.47.

C.9.17 sub_exp_polarity

sub_exp_polarity(+*Exp*, +*ExpPolarity*, +*Pos*, −*SubExpPolarity*)

This predicate calculates the polarity of a subexpression. An expression *Exp* is supplied alongside its known polarity as *ExpPolarity*. A subexpression within *Exp* is indicated via the position *Pos*. Given this information, the predicate calculates the polarity of the subexpression as *SubExpPolarity*.

To minimise implementation effort, polarity is only reported for two tightly constrained situations. Firstly, where the subexpression is the entire input expression, the polarity of the subexpression is reported as being the same as the input expression. Secondly, where the subexpression is a top level conjunct of the input expression, the polarity of the subexpression is reported as being the same as the input expression. In all other cases the polarity is reported as being unknown.

The limited calculation of polarity has the potential to prevent proof. If the polarity of a subexpression is reported as being unknown then polarity dependent rewrite rules may not be applied. In practice, this limitation has not prevented proof, as our proof plans tend to decompose the structure of conclusions such that polarity will eventually be reported. Note that this predicate is responsible for all polarity calculations. Thus, if the limited calculation of polarity became a concern, the proof plans may be enhanced by extending this predicate accordingly.

C.9.18 total_functions

total_functions(+*Exp*, +*OpList*, −*TotalInt*)

This predicate reports the total number of operators *OpList* in expression *Exp* as *TotalInt*. Operators inside program variable expressions, as defined by prog_var_exps, are not counted.

C.9.19 unconstrained_var

unconstrained_var(+*Exp*)

This predicate is successful if expression *Exp* is an unconstrained variable, as described in §G.8.

C.10 Rewriting

C.10.1 constants_to_value

constants_to_value(+*Polarity*, +*InExpList*, −*OutExpList*, −*Tactic*)

This predicate rewrites scalar constants with their known values. The expressions to be rewritten are supplied as *InExpList* alongside their consistent polarity as *Polarity*. Following constant replacements, the rewritten expressions are returned as *OutExpList* alongside a supporting tactic *Tactic* that performs the rewrites at the object-level.

C.10.2 constrain_const_arrays

constrain_const_arrays(−*ConstraintList*, −*Tactic*)

This predicate retrieves constraints for constant arrays that are associated with the plan. The constraints are returned as *ConstraintList* alongside a supporting tactic *Tactic* that introduces the constraints as hypotheses at the object-level.

C.10.3 constrain_exps

constrain_exps(+*HypList*, +*ExpList*,
−*SpecialisedHypList*, −*Tactic*, −*ConditionList*)

This predicate specialises hypotheses *HypList* to constrain expressions of interest *ExpList*. In particular, for an expression X, the predicate attempts to introduce hypotheses of the form $X \geq Y$ and $X \leq Z$, where Y and Z are ground expressions. The specialised hypotheses are returned as *SpecialisedHypList* alongside a tactic *Tactic* that introduces these hypotheses at the object-level. The predicate also returns a list of conditions *ConditionList* whose absence prevented the introduction of specialised hypotheses. These conditions correspond to an implication *Condition* $\rightarrow$ *Property* where *Property* offers a pertinent constraint, yet *Condition* is not known.

C.10.4 eliminate_duplicate_vars

eliminate_duplicate_vars(+*MaxSeqInt*, +*Var*, +*Polarity*, +*Exp*,
−*Tactic*, −*NewExp*)

This predicate is successful where a sequence of rewrites is discovered that eliminates all duplicate occurrences of variable *Var* in expression *Exp* of known polarity *Polarity*. The maximum sequence of rewrites to explore is constrained by *MaxSeqInt*. Where successful, the transformed expression is returned as *NewExp* alongside a tactic *Tactic* that performs the rewrites at the object-level. In applying rewrite rules, the following restrictions are imposed:

- **Ground** - The rewrite rule must not introduce meta-variables, as these would significantly increase the search space.
- **Non-preserving** - The rewrite rule must not entirely preserve the original expression. Such rewrites always increase expression structure, hindering the elimination of variables. For example, the following rewrite is rejected, as the left hand side expression is entirely preserved in the right hand side expression:

$$Z \Rightarrow Z + 0 \tag{C.2}$$

C.10.5 select_alt_view_rule

select_alt_view_rule(+*Polarity*,
−*RewriteForm*,
?*Condition* :?*LHSExp* ⇒?*RHSExp*)

This predicate offers alternative views of an expression by exploring three types of rewrite rule. The expression is supplied through a partially instantiated rewrite rule, whose left hand side has known polarity *Polarity*. Where found, each rewrite rule is described through *RewriteForm*. The three types of rewrite rule sought are listed below:

- **Equal** - Do not modify the expression. For example: $Z \Rightarrow Z$.
- **Commute** - Commute a binary expression. For example: $Y + Z \Rightarrow Z + Y$.
- **Rotate** - Rotate a binary expression. For example: $Y < Z \Rightarrow Z > Y$.

C.10.6 select_rewrite_rule

select_rewrite_rule(+*Polarity*,
−*RewriteForm*,
?*Condition* :?*LHSExp* ⇒?*RHSExp*)

This predicate offers alternative rewrites of an expression by exploring the available rewrite rules. The expression is supplied through a partially instantiated rewrite rule, whose left hand side has known polarity *Polarity*. Where found, each rewrite rule is described through *RewriteForm*.

C.10.7 select_transitivity_rule

select_transitivity_rule(+*Polarity*,
−*RewriteForm*,
?*Condition* :?*LHSExp* ⇒?*RHSExp*)

This predicate offers alternative transitive rewrites of an expression. The expression is supplied through a partially instantiated rewrite rule, whose left hand side has known polarity *Polarity*. Where found, each rewrite rule is described through *RewriteForm*. The transitive rewrites are restricted to those of the following form:

$$Condition : (X\ RelOp\ Y) \Rightarrow (X\ RelOp\ Z) \wedge (Z\ RelOp\ Y) \tag{C.3}$$

Where *X* and *Y* are expressions, *RelOp* is an inequality relation, and *Z* is an introduced meta-variable.

C.11 Rippling

C.11.1 ripple_annotate

ripple_annotate(+*IndHyp*, +*IndConc*, −*AnnIndConc*) ©

This predicate is successful where induction conclusion *IndConc* annotated with respect to its induction hypotheses *IndHyp* leads to the annotated induction conclusion *AnnIndConc*.

C.11.2 ripple_complete

ripple_complete(+*AnnIndConc*, −*IndHyp*, −*IndConc*, −*IndHypPos*) ©

This predicate is successful where the annotated induction conclusion *AnnIndConc* is fully rippled. In this case, the unannotated induction hypothesis and induction conclusion is reported as *IndHyp* and *IndConc* respectively. Further, the position of the induction hypothesis within the induction conclusion is reported as *IndHypPos*.

C.11.3 ripple_erasure

ripple_erasure(+*AnnExp*, −*Exp*) ©

This predicate erases annotations from annotated expression *AnnExp* returning the result as *Exp*.

C.11.4 ripple_exp_at

ripple_exp_at(+*Exp*, −*Pos*, −*SubExp*) ©

This predicate returns every well-annotated subexpression of *Exp* as *SubExp* alongside its position *Pos*.

C.11.5 ripple_unblock_strategies

ripple_unblock_strategies(+*AnnIndConc*, −*UnblockedAnnIndConc*, −*Tactic*) ©

This predicate attempts to unblock a ripple proof step by transforming the annotated induction conclusion *AnnIndConc*. Where successful, the modified induction conclusion is returned as *UnblockedAnnIndConc* alongside a tactic *Tactic* that performs the transformation at the object-level. For further details on the unblocking strategies considered, see the ripple_unblock method in §E.38.

C.11.6 select_wave_rule

select_wave_rule(+*Polarity*, ©
−*RewriteForm*,
?*Condition* :?*LHSExp* ⇒?*RHSExp*)

This predicate offers alternative rewrites of an expression via the available wave-rules. The expression is supplied through a partially instantiated rewrite rule, whose left hand side has known polarity *Polarity*. Where found, each rewrite rule is described through *RewriteForm*.

Appendix D

Tacticals and Tactics

D.1 Introduction

Proof planners typically check the correctness of discovered proof plans in a tactic based theorem prover. To integrate with the SPARK Approach, our proof plans are checked in the Checker, which is not a tactic based theorem prover. To address the mismatch, a collection of simulated tactics and tacticals are introduced. Each of the simulated tactics and tacticals, including their translation into Checker commands, are listed in this chapter. Note that the translation is supported though richer proof commands introduced through modifications made to the Checker, as detailed in §B.4.3.

D.2 Tactics

D.2.1 null_tactic

null_tactic

This tactic does nothing, making zero changes to the object-level goal. For Checker translation, the tactic is simply ignored.

D.2.2 trivial_tactic

trivial_tactic

This tactic appeals to the automated reasoning capabilities of the Checker to discharge a goal.

Checker Translation
`tame_done.`

D.2.3 trivially_true_conc_tactic

trivially_true_conc_tactic(*Conc*)

This tactic appeals to the automated reasoning capabilities of the Checker to replace conclusion *Conc* with *true*.

Checker Translation
`tame_subgoal_on_exp (`*Conc*`).`
`tame_done.`
`tame_all_done.`
`tame_rewrite conc : `*Conc*` : [] where `*Conc*`.`

D.2.4 rewrite_tactic

rewrite_tactic(*RewriteForm*,
<*HypOrConc*>, *WholeExp*, *Pos*,
Condition : *LHSExp* ⇒ *RHSExp*)

This tactic rewrites an expression in the goal. The tactic receives a collection of structured arguments, as shown in Figure D.1. The form of the rewrite is described through *RewriteForm* as summarised below:

- *rule*(*FileName*, <*FileKind*>, *RuleId*, <*Direction*>) - Rewrite an expression by applying an external rewrite rule. The rewrite rule is referenced through its file *FileName*, file kind <*FileKind*> and rule identifier *RuleId*. The rule is applied in direction <*Direction*>.
- *hypothesisRewrite*(*HypRewriteExp*, <*Direction*>) - Rewrite an expression by treating hypothesis *HypRewriteExp* as a rewrite rule, in direction <*Direction*>.
- *hypothesisFertilise*(*HypFertiliseExp*) - Rewrite an expression as *true* via matching hypothesis *HypFertiliseExp*.
- *evaluate*(*EvaluatableExp*, *Value*) - Rewrite an expression *EvaluatableExp* to the result of its evaluation *Value*.

The application of the rewrite is described through a combination of items, as summarised below:

- <*HypOrConc*> - Indicates whether a hypothesis or conclusion is to be rewritten as *hyp* or *conc* respectively.
- *WholeExp* - The whole expression of the hypothesis or conclusion to be rewritten.
- *Pos* - Selects the position within the hypothesis or conclusion that is to be rewritten.
- *Condition* : *LHSExp* ⇒ *RHSExp* - Some rewrite rules admit the introduction of new structure through meta-variables. The new structure is selected by specifying the concrete form of the rewrite rule.

<*RewriteForm*> ::= *rule*(*FileName*, <*FileKind*>, *RuleId*, <*Direction*>) |
hypothesisRewrite(*HypRewriteExp*, <*Direction*>) |
hypothesisFertilise(*HypFertiliseExp*) |
evaluate(*EvaluatableExp*, *Value*)
<*Direction*> ::= *normal* | *reversed*
<*FileKind*> ::= | *rul* | *rlu* | *rls*
<*HypOrConc*> ::= *hyp* | *conc*

Figure D.1: Arguments for rewrite_tactic

Checker Translation for: *rule*(*FileName*, <*FileKind*>, *RuleId*, <*Direction*>)

If <FileKind> is rlu or rls the rule is not available by default. Such rule files must be explicitly loaded before they are used.
`consult` *FileName* `.`
`tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `with` *LHSExp* `to` *RHSExp*
`if` *Condition* `using` *RuleId* `in` <*Direction*>`.`

Checker Translation for: *hypothesisRewrite*(*HypRewriteExp*, <*Direction*>)
`tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `with` *LHSExp* `to` *RHSExp* `if` *Condition* `from` *HypRewriteExp* `in` <*Direction*>`.`
Checker Translation for: *hypothesisFertilise*(*HypFertiliseExp*)
`tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `where` *HypFertiliseExp*`.`
Checker Translation for: *evaluate*(*EvaluatableExp*, *Value*)
`tame_rewrite` *HypOrConc* `:` *WholeExp* `:` *Pos* `with` *EvaluatableExp* `is` *Value*`.`

D.2.5 split_conc_conj_tactic

split_conc_conj_tactic(*LeftExp*, *RightExp*)

This tactic splits a goal with a conjoined conclusion *LeftExp* ∧ *RightExp* into subgoals with conclusions *LeftExp* and *RightExp*.

Checker Translation

`tame_subgoal_on_exp (`*LeftExp*`).`
Execute the tactics targeted at the first (left) subgoal.
`tame_all_done.`
`tame_subgoal_on_exp (`*RightExp*`).`
Execute the tactics targeted at the second (right) subgoal.
`tame_all_done.`
`tame_done.`

D.2.6 case_split_tactic

case_split_tactic(*FirstExp*, *SecondExp*)

This tactic performs a case split based on a disjoined property *FirstExp* ∨ *SecondExp*. Each case is considered as a separate subgoal, extended to include either the additional hypothesis *FirstExp* or *SecondExp*.

Checker Translation
`prove c#1 by cases on (`*FirstExp* `or` *SecondExp*`).`
Execute the tactics targeted at the first case.
`tame_all_done.`
Execute the tactics targeted at the second case.
`tame_all_done.`
`tame_done.`

D.2.7 sequence_tactic

sequence_tactic(*TacticList*)

This tactic executes a list of tactics in sequence, provided as *TacticList*. For Checker translation, each tactic of *TacticList* is translated in sequence.

D.3 Tacticals

D.3.1 then_tactical

then_tactical(*Tactic*, *TacticalList*)

This tactical applies tactic *Tactic* to the goal, generating *n* subgoals. Following this, the i^{th} tactical in the list *TacticalList* is applied to the i^{th} subgoal. For Checker translation, first *Tactic* is translated, followed by an ordered translation of the tacticals in *TacticalList*.

D.3.2 final_tactical

final_tactical(*Tactic*)

This tactical applies tactic *Tactic* to the goal. The expectation is that the tactic will discharge the goal, leaving no subgoals. For Checker translation *Tactic* is translated.

Appendix E

Proof Plans

E.1 Introduction

As described in §6.7, our proof plans are expressed through methods and critics and the application of methods is controlled through strategies. Each of these components are detailed in this chapter. Proof plans for exception freedom goals are introduced in §E.2 while proof plans for program analysis queries are introduced in §E.39. The method-language supporting the expression of methods and critics is detailed in Appendix C.

E.2 Proof Plans for Exception Freedom Goals

Proof plans are developed for exception freedom goals that arise in the SPARK Approach, as detailed in the following sections.

E.3 Strategy: exception_freedom

The exception_freedom strategy is shown in Figure E.1 and described below.

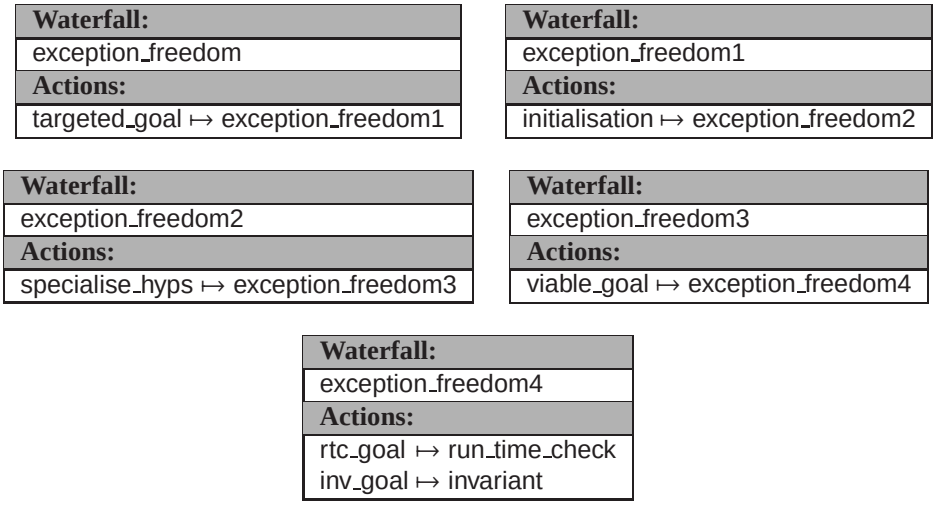


Figure E.1: exception_freedom strategy

E.3.1 Behaviour

This is the entry strategy for proving exception freedom goals. The strategy targets those goals that have not been proved by the Simplifier. Further, the strategy targets the initial form of goals, rather than their simplified form. The targeted goals are refined through an initialisation process. Prior to attempting proof, each goal is investigated to determine its viability. Those goals that appear to be provable are explored further. Different strategies are selected for run-time check goals and invariant goals, appealing to their different characteristics.

E.4 Strategy: run_time_check

The run_time_check strategy is shown in Figure E.2 and described below.

Waterfall:	
run_time_check	
Actions:	
true_conc	↦ ∅
false_conc	↦ run_time_check
linear_bounded_conc	↦ run_time_check
case_split	↦ run_time_check
mult_commute	↦ run_time_check
eval_conc	↦ run_time_check
split_conc_conj	↦ run_time_check
fertilize	↦ run_time_check
clear_conc_exp	↦ run_time_check
elim_var_conc	↦ run_time_check
transitivity_entry	↦ transitivity(run_time_check)

Waterfall:	
transitivity(*ContinuationStrategy*)	
Actions:	
transitivity_fertilize	↦ transitivity
transitivity_decomp	↦ transitivity
transitivity_close	↦ *ContinuationStrategy*
transitivity_unblock	↦ transitivity

Figure E.2: run_time_check strategy

E.4.1 Behaviour

This strategy proves run-time check goals. The strategy is also used to prove lemmas and subgoals that emerge in proving invariant goals. The strategy considers increasingly sophisticated methods to advance proof. In particular, as a last resort, the strategy seeks to decompose a conclusion by introducing a transitivity step.

E.5 Strategy: invariant

The invariant strategy is shown in Figure E.3 and described below.

Waterfall:
invariant
Actions:
ripple_entry ↦ ripple(run_time_check)

Waterfall:
ripple(*ContinuationStrategy*)
Actions:
ripple_unblock ↦ ripple
ripple_wave ↦ ripple
ripple_fertilize ↦ *ContinuationStrategy*

Figure E.3: invariant strategy

E.5.1 Behaviour

This strategy proves invariant goals. The strategy immediately attempts to introduce a ripple step.

E.6 Method: targeted_goal

The targeted_goal method is shown in Figure E.4 and described below. The four critics associated with this method are proved_at_simplifier, simplified_goal, other_goal and in_real_domain as described in §E.7, §E.8, §E.9 and §E.10 respectively.

Method:
targeted_goal
Tactic:
null_tactic
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Not proved by Simplifier.
match_global_context(*provedAtSimplifier*(*false*))
Not a simplified goal.
match_global_context(*sourceSystem*(*vcg*))
Goal is of a targeted category.
not(get_goal_category(*other*)),
No real arithmetic in the conclusion.
remove_real_exps([*Conc*], [*Conc*])
Effects:
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *Conc*]

Figure E.4: targeted_goal method

E.6.1 Behaviour

This method is successful where the goal is targeted by our proof plans. Four separate checks are performed as listed below:

- **Not already proved** - The goal has not been proved by the Simplifier.
- **Not a simplified goal** - Each goal not proved by the Simplifier will be encountered in both its initial and simplified form. The simplified goals are subject to significant and variable structural changes, making it difficult to identify proof families. In particular, the structural changes can prevent the introduction of a ripple step. Thus, the initial form of goals are targeted.
- **Is a targeted goal** - Exception freedom goals and their related invariant goals are targeted. It is unlikely that progress will be made for other goal categories, so they are not considered.
- **Conclusion not in the real domain** - As discussed in §6.5.1, our proof plans target discrete types. The check is only performed on the conclusion, supporting the investigation of goals that have discrete conclusions with some hypotheses in the real domain.

E.7 Critic: proved_at_simplifier

The proved_at_simplifier critic is shown in Figure E.5 and described below. The critic is associated with the targeted_goal method, as described in §E.6.

Critic:
proved_at_simplifier
Parent method:
targeted_goal
Goal:
_
Successful method preconditions:
∅
Failed method precondition:
match_global_context(*provedAtSimplifier*(*false*))
Preconditions:
∅
Effects:
write_line('Goal already proved by the Simplifier.') abort_plan(*provedBySimplifier*)

Figure E.5: proved_at_simplifier critic

E.7.1 Behaviour

Where the goal has been proved by the Simplifier the targeted_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique.

E.8 Critic: simplified_goal

The simplified_goal critic is shown in Figure E.6 and described below. The critic is associated with the targeted_goal method, as described in §E.6.

Critic:
simplified_goal
Parent method:
targeted_goal
Goal:
_
Successful method preconditions:
match_global_context(*provedAtSimplifier*(*false*))
Failed method precondition:
match_global_context(*sourceSystem*(*vcg*))
Preconditions:
∅
Effects:
write_line('Is a simplified goal.') abort_plan(*simplifiedGoal*)

Figure E.6: simplified_goal critic

E.8.1 Behaviour

Where the structure of the goal has been subject to simplification the targeted_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique.

E.9 Critic: other_goal

The other_goal critic is shown in Figure E.7 and described below. This critic is associated with the targeted_goal method, as discussed in §E.6.

Critic:
other_goal
Parent method:
targeted_goal
Goal:
_
Successful method preconditions:
match_global_context(*provedAtSimplifier*(*false*)) match_global_context(*sourceSystem*(*vcg*))
Failed method precondition:
not(get_goal_category(*other*))
Preconditions:
∅
Effects:
write_line('Is not a targeted goal.') abort_plan(*goalNotTargeted*)

Figure E.7: other_goal critic

E.9.1 Behaviour

Where the goal category is not targeted by the proof plans, the targeted_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique.

E.10 Critic: in_real_domain

The in_real_domain critic is shown in Figure E.8 and described below. This critic is associated with the targeted_goal method, as discussed in §E.6.

Critic:
in_real_domain
Parent method:
targeted_goal
Goal:
_
Successful method preconditions:
match_global_context(*provedAtSimplifier*(*false*)) match_global_context(*sourceSystem*(*vcg*)) not(get_goal_category(*other*))
Failed method precondition:
remove_real_exps([*Conc*], [*Conc*])
Preconditions:
∅
Effects:
write_line('Conclusion in real domain.') abort_plan(*inRealDomain*)

Figure E.8: in_real_domain critic

E.10.1 Behaviour

Where the goal has a conclusion with expressions in the real domain, the targeted_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique.

E.11 Method: initialisation

The initialisation method is shown in Figure E.9 and described below.

Method:
initialisation
Tactic:
sequence_tactic([*ConstraintTactic*, *HypTactic*, *ConcTactic*])
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
∅
Effects:
Find constraints for constant arrays.
constrain_const_arrays(*ConstraintList*, *ConstraintTactic*)
Add the constraints as hypotheses.
append(*HypList*, *ConstraintList*, *ExtendedHypList*)
Remove duplicate hypotheses.
filter_duplicates(*ExtendedHypList*, _, *UniqHypList*)
Remove real hypotheses.
remove_real_exps(*UniqHypList*, *DiscreteHypList*)
Replace named scalar constants for all hypotheses.
constants_to_value(*negative*, *DiscreteHypList*, *NewHypList*, *HypTactic*)
Replace named scalar constants for the conclusion.
constants_to_value(*positive*, [*Conc*], [*NewConc*], *ConcTactic*)
Subgoals:
[*LocalContextList* : *NewHypList* ⊢ *NewConc*]

Figure E.9: initialisation method

E.11.1 Behaviour

The Examiner is a strictly mechanical verification condition generator. Consequently, the goals encountered tend to be relatively verbose. This method performs four separate initialisations, streamlining goals to ease proof.

E.11.2 Introduce External Constraints

The Examiner does not generate subprogram rules associating constant arrays with their corresponding constant expressions. This behaviour is selected as the generation of large constant arrays could adversely affect the performance of the toolset. Instead, an engineer may introduce user rules to suitability constrain constant arrays. Where present, these rules are directly relevant to the goal. This method identifies such rules and introduces them as hypotheses.

Note that recent versions of the Examiner can generate subprogram rules associating constant arrays with their corresponding constant expressions. To address performance concerns, the generation of these rules is configurable.

E.11.3 Remove Duplicate Hypotheses

The goal may contain duplicate hypotheses. Such hypotheses often arise from the same variable being used in different contexts, generating duplicate occurrences of its type constraints. This method identifies and removes duplicate hypotheses. The removal of duplicate hypotheses has no effect on proof as each remaining hypothesis can be used wherever its duplicates may have been used. Consequently, the remove of duplicate hypotheses at the meta-level need not be performed at the object-level.

E.11.4 Remove Real Hypotheses

The targeted_goal method ensures that the conclusion is not in the real domain. However, there may be hypotheses in the real domain. As our plans target the discrete domain, it is unlikely that these hypotheses will be required. Thus, this method identifies and removes any hypotheses in the real domain. The presence of unused hypotheses has no effect on proof. Consequently, the removal of real hypotheses at the meta-level need not be performed at the object-level.

E.11.5 Replace Named Scalar Constants With Their Values

The Examiner generates subprogram rules, associating scalar constants with their corresponding constant expressions. Two alternative techniques were considered for exploiting these rules:

- **Unconstrained** - Replace every named scalar constant with its corresponding value. This is a trivial operation that can be performed in a single method. However, inevitability, unnecessary constant replacements will occur, resulting in a less succinct proof.

- **Constrained** - Only replace named scalar constants with their corresponding value if this replacement is necessary to complete a proof. Middle-out reasoning might be used to discover how the proof will progress and identify targeted constant replacements. Such an approach would be non-trivial, requiring appropriate communication between methods. However, the resulting proofs will be more succinct.

Replacing named scalar constants with their corresponding values is regarded as an obvious simplification, rather than a key step of proof development. On this basis, unconstrained replacement was adopted.

A ripple step is dependent on finding structural matches between a hypothesis and conclusion. Changes to the syntax of a goal has the potential to disrupt a ripple proof. Here, each occurrence of a named constant will be universally replaced with the same value. Thus, the number of structural matches will not reduce. However, the number of structural matches might increase, if previously distinct named constants are replaced

with the same value. In principle, this might create additional search in developing a ripple step. However, in practise, the situation has not been encountered.

E.12 Method: specialise_hyps

The specialise_hyps method is shown in Figure E.10 and described below.

Method:
specialise_hyps
Tactic:
sequence_tactic([*LemmaTactic*, *AdditionalTactic*])
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
∅
Effects:
Collect program variable expressions in the conclusion.
prog_var_exps(*Conc*, *ProgVarExpList*_)
Identify supplementary lemmas for extending hypotheses.
constrain_exps(*HypList*, *ProgVarExpList*, _, _, *LemmaList*)
Try to prove each supplementary lemma.
plan_lemmas(*HypList*, *LemmaList*, run_time_check, *LemmaProvedList*, *LemmaTactic*)
Extend hypotheses to include the lemmas.
append(*HypList*, *LemmaProvedList*, *ExtendedHypList*)
Identify additional hypotheses.
constrain_exps(*ExtendedHypList*, *ProgVarExpList*, *AdditionalHypList*, *AdditionalTactic*, _)
Get extended hypotheses.
append(*ExtendedHypList*, *AdditionalHypList*, *NewHypList*)
Subgoals:
[*LocalContextList* : *NewHypList* ⊢ *Conc*]

Figure E.10: specialise_hyps method

E.12.1 Behaviour

Typically, hypotheses offer general constraints while a conclusion requires demonstrating a specific constraint. For example, a hypothesis may constrain every element of an array to be within its type while a conclusion may require demonstrating that a particular element of this array is within its type. In such cases, proof often involves specialising general hypothesis constraints to target the specific conclusion constraints. Two alternative techniques were considered for achieving this hypothesis specialisation:

- **Unconstrained** - Preemptively specialise hypotheses to target the specific form of the conclusion. In general, it is difficult to predict the structure of a proof and hence difficult to determine which specialised hypotheses will be required. However, in verifying exception freedom, the structure of the conclusion can offer strong guidance in selecting relevant hypothesis specialisations.

- **Constrained** - Only specialise hypotheses where this is strictly necessary to complete proof. Middle-out reasoning might be used to discover how the proof will progress and identify targeted hypothesis specialisations. The approach would naturally support the discovery and introduction of hypothesis specialisations that are not intuitively suggested by the conclusion. However, several methods would be affected by this approach as access to relevant hypotheses is a common requirement.

The constrained technique offers a powerful and targeted mechanism for hypothesis specialisation. However, the unconstrained technique is significantly simpler and, in verifying exception freedom, sufficiently effective. On this basis, unconstrained hypothesis specialisation was adopted. Note that, if extending these plans further, the unconstrained technique might be complemented through the introduction of the constrained technique. This might be expressed as a critic, offering insightful hypothesis specialisations to patch otherwise failing subgoals.

E.12.2 Plan Lemmas Separately

The specialisation of hypotheses may require proving supplementary lemmas. For example, to specialise a hypothesis constraining every element of an array to a hypothesis constraining a particular element of an array it must be shown that the particular element lies within the range of the array. As described in §6.8.2, our proof planner supports the simultaneous development of multiple plans. This mechanism is exploited to plan lemmas separately. The style is advantages as it supports the reuse of existing strategies. Typically, each lemma requires proving that a particular expression lies inside a general range. There is a strong correspondence between this task and proving run-time check goals. Thus, the run_time_check strategy is reused in planning lemmas.

E.13 Method: viable_goal

The viable_goal method is shown in Figure E.11 and described below. The four critics associated with this method are couple_entry_vars, constrain_consts, constrain_vars and tightly_constrain_vars as described in §E.14, §E.15, §E.16 and §E.17 respectively.

Method:
viable_goal
Tactic:
null_tactic
Goal:
Goal
Preconditions:
∅
Effects:
No uncoupled entry variables.
uncoupled_entry_vars(*Goal*, *UncoupledVarList*) *UncoupledVarList* = []
Check no unconstrained constants or variables.
unconstrained_consts_vars(*Goal*, *UnconstrainedConstList*, *UnconstrainedVarList*) *UnconstrainedConstList* = [] *UnconstrainedVarList* = []
No under constrained variables.
under_constrained_vars(*Goal*, *UnderConstrainedVarList*) *UnderConstrainedVarList* = []
Subgoals:
[*Goal*]

Figure E.11: viable_goal method

E.13.1 Behaviour

This method searches for goal patterns associated with unprovable goals. The method is only successful where none of these patterns occur. As such, the method effectively expresses preconditions for an entire strategy. Three difficult patterns of unprovable goal are considered, as described in the sections below.

E.13.2 No Uncoupled Entry Variables

A SPARK for-loop terminates when its iterator variable reaches an end-point value. The end-point value is calculated by evaluating the end-point expression as the loop is *entered*. Typically, a for-loop will iterate over a subtype. In this case the end-point value is simply the last value of this subtype. However, a for-loop may have a more complex end-point expression, referencing program variables. Significantly, these variables may be modified within the loop. Thus, the evaluation of the end-point expression at loop entry may differ from its evaluation on subsequent iterations. To capture these semantics in program verification, every variable in an end-point expression is cloned as a special *entry* variable.

Each entry variable takes the value of its corresponding variable at the point the loop is entered. Thus, the evaluation of an end-point expression, in terms of the entry variables, is the same on every loop iteration. This transformed end-point expression is used to describe the end-point value of the for-loop.

The entry variable mechanism faithfully represents the semantics of for-loops. However, where present, entry variables typically become an obstacle to proof. It is nearly always necessary to introduce an invariant that describes the relationship between each entry variable and its corresponding program variable. Thus, this method rejects goals that are missing such missing invariants.

```
package WriteToArrayPartition_Package is
  subtype I_Type is Integer range 0 .. 100;
  type D_Type is array (I_Type) of Integer;
  procedure WriteToArrayPartition(Left: in I_Type;
                                  Right: in I_Type;
                                  Value: in Integer;
                                  Destination: in out D_Type);
  --# derives Destination from Left, Right, Value, Destination;
end WriteToArrayPartition_Package;
```

```
package body WriteToArrayPartition_Package is
  procedure WriteToArrayPartition(Left: in I_Type;
                                  Right: in I_Type;
                                  Value: in Integer;
                                  Destination: in out D_Type)
  is
  begin
    for I in I_Type range Left .. Right loop
      --# assert true;
      Destination(I):=Value;
    end loop;
  end WriteToArrayPartition;
end WriteToArrayPartition_Package;
```

Figure E.12: WriteToArrayPartition subprogram

For example, consider the WriteToArrayPartition subprogram shown in Figure E.12. The subprogram writes a given value to a bounded portion of an array. The subprogram contains a for-loop that terminates when the loop iterator *i* reaches the value of variable *right* at loop entry. The essential goal[1] for verifying that *i* does not exceed its upper bound

[1]In the SPARK Approach, the entry variable for *right* is referenced as *right__entry__loop__<counter>*. This verbose name is guaranteed to be unique within its enclosing subprogram. For brevity, in the examples shown in this thesis, every entry variable is uniquely referenced via its program variable name, appended with *_entry*.

within the for-loop, following the initialisation method, is shown below:

$$\begin{array}{l}(i \leq 100) \wedge (right \leq 100) \wedge (i \leq right\_entry) \wedge (\neg(i = right\_entry)) \\ \rightarrow \\ (i + 1) \leq 100\end{array} \tag{E.1}$$

The goal is unprovable as there is no hypothesis relating the entry variable *right_entry* to the program variable *right*. As variable *right* is not modified within the for-loop an invariant could be introduced equating *right_entry* with *right*. With such an invariant in place, it would become possible to prove the goal.

Recent versions of the Examiner are more selective in the introduction of entry variables. All import variables of mode *in* cannot be modified within a subprogram. Thus, the entry value of these variables will always equal their corresponding program variable. On this basis, for such variables, the Examiner omits the introduction of entry variables.

E.13.3 No Unconstrained Constants or Variables

Following the initialisation method, every available constant constraint, in either the subprogram or user rules, will have been introduced. As standard, every variable in the goal should be constrained to be within its type. Thus, any unconstrained constants or variables strongly indicate that the goal needs strengthening. Consequently, this method rejects goals where unconstrained constants or variables can be identified. Note that an exception is made for entry variables, as these are often constrained indirectly, by being related to their corresponding program variable.

E.13.4 No Under Constrained Constants or Variables

A constraint solver is exploited to search for a counter-example to the goal. The counter-example identifies a collection of constants and variables whose constraints are likely to need strengthening in order to prove the goal. The technique is detailed in the sections below, and illustrated through the FilterShortInteger subprogram shown in Figure E.13. The subprogram sums the values of an array that lie between 0 and 100.

Constraint Solver

Constraint solving can be a computationally intensive task. To make this task tractable, most constraint solvers operate in restricted domains. We focus on the clp(FD) (Constraint Logic Programming Finite Domain) constraint solver [COC97], which is distributed with Sicstus Prolog [Swe05]. The clp(FD) constraint solver operates with integers that lie between $-(2^{25})$ and $(2^{25}) - 1$. Further, the default configuration of the clp(FD) constraint solver supports a relatively limited number of functions, as shown by its grammar in

```
package FilterShortInteger_Package is
  subtype AR_T is Short_Integer range 0..9;
  type A_T is array (AR_T) of Short_Integer;
  procedure FilterShortInteger(A: in A_T; R: out Short_Integer);
  --# derives R from A;
end FilterShortInteger_Package;
```

```
package body FilterShortInteger_Package is
  procedure FilterShortInteger(A: in A_T; R: out Short_Integer)
  is
  begin
    R:=0;
    for I in AR_T loop
      --# assert true;
      if A(I)>=0 and A(I)<=100 then
        R:=R+A(I);
      end if;
    end loop;
  end FilterShortInteger;
end FilterShortInteger_Package;
```

Figure E.13: FilterShortInteger subprogram

Figure E.14. Nevertheless, within this restricted domain, the constraint solver is capable of performing sophisticated reasoning in a timely fashion.

<*Constraint*> ::= <*Constraint*> $\wedge$ <*Constraint*> |
<*Constraint*> $\vee$ <*Constraint*> |
$\neg$<*Constraint*> |
<*Eq*> |
<*boolean-variable*>

<*Eq*> ::= <*IntExp*> = <*IntExp*> | <*IntExp*> $\neq$ <*IntExp*> |
<*IntExp*> < <*IntExp*> | <*IntExp*> $\leq$ <*IntExp*> |
<*IntExp*> > <*IntExp*> | <*IntExp*> $\geq$ <*IntExp*>

<*IntExp*> ::= <*IntExp*> $*$ <*IntExp*> |
<*IntExp*> *div* <*IntExp*> | <*IntExp*> *mod* <*IntExp*> |
<*IntExp*> + <*IntExp*> | <*IntExp*> − <*IntExp*> | −<*IntExp*> |
<*integer-variable*> |
<*integer*> {in range: $-(2^{25}) \ldots (2^{25}) - 1$}

Figure E.14: clp(FD) input grammar

Reject Goals Outside Integer Domain

When the constraint solver encounters integers that lie beyond its legal range an overflow error is raised and the constraint solving request is abandoned. The situation will occur if the input constraint problem contains integers outside the legal range. Further, the situation will occur if, during constraint solving, calculations are performed that generate integers outside the legal range.

To guard against overflows, constraint solving is only attempted where every constant lies well within the legal range accepted by the constraint solver. In practice, it is checked that every constant lies between the bounds $-(2^{20})$ and $(2^{20}) - 1$. While this restriction is simplistic, it offers a reasonable assurance that an overflow will not occur during constraint solving. For example, in the FilterShortInteger subprogram, every value is represented as a short integer. As specified in the target configuration file of §4.4.5, these integers are bound between $-(2^{15})$ and $(2^{15}) - 1$. Significantly, these bounds lie inside the range considered by this method.

Negate Goal to Search for Counter-Example

The aim is to identify situations where the goal is false. This is achieved by searching for solutions that satisfy the negation of the goal. In general, each input goal takes the form:

$$\forall vars.\ ((Hyp_1 \wedge \cdots \wedge Hyp_n) \rightarrow Conc)$$

Negating and simplifying this goal leads to:

$$\exists vars.\ ((Hyp_1 \wedge \cdots \wedge Hyp_n) \wedge (\neg Conc))$$

Significantly, in negating the goal, each variable is transformed from being implicitly universally quantified to implicitly existentially quantified. For example, consider the FilterShortInteger subprogram. In verifying exception freedom, it must be shown that the assignment `R:=R+A(I)` always assigns a value to r that is within its upper bound. The corresponding goal and its negation, following the initialisation method, is shown in Figure E.15.

Overflow goal
$\forall(i\_1 : short\_integer.\ i\_1 \geq 0 \wedge i\_1 \leq 9 \rightarrow$ $element(a,[i\_1]) \geq -32768 \wedge element(a,[i\_1]) \leq 32767) \wedge$ $(i \geq 0) \wedge (i \leq 9) \wedge (r \geq -32768) \wedge (r \leq 32767) \wedge$ $(element(a,[i]) \geq 0) \wedge (element(a,[i]) \leq 100)$ $\rightarrow$ $r + element(a,[i]) \leq 32767$

Negated overflow goal
$\forall(i\_1 : short\_integer.\ i\_1 \geq 0 \wedge i\_1 \leq 9 \rightarrow$ $element(a,[i\_1]) \geq -32768 \wedge element(a,[i\_1]) \leq 32767) \wedge$ $(i \geq 0) \wedge (i \leq 9) \wedge (r \geq -32768) \wedge (r \leq 32767) \wedge$ $(element(a,[i]) \geq 0) \wedge (element(a,[i]) \leq 100) \wedge$ $\neg(r + element(a,[i]) \leq 32767)$

Figure E.15: Overflow goal and its negation

Partition Goal to Meet Input Grammar

It is unlikely the negated goal will reside entirely within the constraint solver grammar. To address this, the goal is partitioned into three separate conjunct lists as summarised below:

- *within* - Conjuncts within the constraint solver grammar.
- *Beyond* - Conjuncts beyond the constraint solver grammar.
- *equalities* - Conjuncts equating integer expressions to integer variables.

Each conjunct of the negated goal is partitioned through the following operations, considered in order:

- **Add to *within*** - The conjunct can be directly expressed in the constraint solver grammar. The conjunct is added to *within*.
- **Eliminate integer expression and repeat** - The conjunct contains a blocking integer expression that can not be directly expressed in the constraint solver grammar. The *equalities* are extended, introducing an equality between the blocking expression and a new integer variable. The blocking expression is then eliminated by being replaced with its corresponding integer variable. For consistency, the elimination is preformed throughout the negated goal and the emerging partitioned goal in *within* and *beyond*. With the blocking expression eliminated, the partitioning process is repeated.
- **Add to *beyond*** - Neither of the above cases are applicable. The conjunct is added to *beyond*. Where the negated conclusion can not be presented to the constraint solver it is likely that flawed counter-examples will be discovered. Thus, in this case, the constraint solving attempt is abandoned.

For example, consider the negated goal shown in Figure E.15. Partitioning this goal for constraint solving generates the vales of *within*, *beyond* and *equalities* as shown in Figure E.16. Note that, during partitioning, the integer variable *iv* is introduced to eliminate the integer expression $element(a, [i])$.

Solve Goal as Constraint Problem

After partitioning, the negated goal is expressed through *within*, *beyond*, and *equalities*. The conjuncts in *within* are sent to the constraint solver. Where successful, the constraint solver will discover at least one satisfying solution as *solution*. For example, consider the *within* partition show in Figure E.16. The first satisfiable solution discovered is:

$$(i = 0) \wedge (r = 32668) \wedge (iv = 100) \tag{E.2}$$

within
$(i \geq 0) \wedge (i \leq 9) \wedge (r \geq -32768) \wedge (r \leq 32767) \wedge (iv \geq 0) \wedge (iv \leq 100) \wedge$ $\neg(r + iv \leq 32767)$

beyond
$\forall(i\_1 : short\_integer.\ i\_1 \geq 0 \wedge i\_1 \leq 9 \rightarrow$ $element(a, [i\_1]) \geq -32768 \wedge element(a, [i\_1]) \leq 32767)$

equalities
$element(a, [i]) = iv$

Figure E.16: Partitioned negated goal

Assemble Candidate Solution

Each *solution* represents a candidate instantiation of existentially quantified variables in the negated goal. On this basis, the negated goal is reassembled for analysis. The *within* conjuncts are replaced by *solution*, and the *equalities* are eliminated by replacing all integer variables with their corresponding expressions. For example, reassembling the partitioned negated goal of Figure E.16, exploiting solution (E.2), leads to the candidate solution shown in Figure E.17.

$(i = 0) \wedge (r = 32668) \wedge (element(a, [i]) = 100) \wedge$ $\forall(i\_1 : short\_integer.\ i\_1 \geq 0 \wedge i\_1 \leq 9 \rightarrow$ $element(a, [i\_1]) \geq -32768 \wedge element(a, [i\_1]) \leq 32767)$

Figure E.17: Candidate solution

Identify Under Constrained Variables

This method assumes that the first candidate solution is valid. Such an assumption is unsound, as only a portion of the goal is submitted to the constraint solver. In principle, this may result in the method falsely reporting under constrained variables. In practice, where verifying exception freedom, this is thought unlikely. Many of the expressions related to exception freedom goals can be expressed in the constraint solver grammar. In particular, constraint solving is only attempted where the negated conclusion can be submitted to the constraint solver. Thus, typically, a significant portion of the goal is submitted to the constraint solver, giving generally strong results.

The method might be strengthened to consider each candidate solution, and attempt to prove that the goal is satisfiable. As the constraint solver may report many solutions, domain knowledge might be exploited to target more promising solutions. For example, in verifying exception freedom, the extreme upper and lower limits of variables are more likely to correspond to genuine counter-examples.

Where a candidate solution is discovered, the method rejects the goal and the variables in *solution* are reported as being under constrained. Entry variables are omitted, as these are constrained indirectly. For example, based on solution (E.2), the for-loop variable *i*, integer variable *r* and array *a* are reported as being under constrained.

E.14 Critic: couple_entry_vars

The couple_entry_vars critic is shown in Figure E.18 and described below. This critic is associated with the viable_goal method, as discussed in §E.13.

Critic:
couple_entry_vars
Parent method:
viable_goal
Goal:
Goal
Successful method preconditions:
uncoupled_entry_vars(*Goal*, *UncoupledVarList*)
Failed method precondition:
UncoupledVarList = []
Preconditions:
∅
Effects:
match_global_context(*sourceSubprogram*(*SubprogramName*)) write_line('Uncoupled entry variable(s) detected.') abort_plan(*abstractPredicate*(*SubprogramName*, *coupleWithEntryVars*(*UncoupledVarList*)))

Figure E.18: couple_entry_vars critic

E.14.1 Behaviour

Where the goal does not contain a hypothesis relating an entry variable to its corresponding program variable the viable_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique, ultimately triggering program analysis.

E.15 Critic: constrain_consts

The constrain_consts critic is shown in Figure E.19 and described below. This critic is associated with the viable_goal method, as discussed in §E.13.

Critic:
constrain_consts
Parent method:
viable_goal
Goal:
Goal
Successful method preconditions:
uncoupled_entry_vars(*Goal*, *UncoupledVarList*) *UncoupledVarList* = [] unconstrained_consts_vars(*Goal*, *UnconstrainedConstList*, *UnconstrainedVarList*)
Failed method precondition:
UnconstrainedConstList = []
Preconditions:
∅
Effects:
match_global_context(*sourceSubprogram*(*SubprogramName*)) write_line('Under constrained constant(s) detected.') abort_plan(*interactionNeeded*(*SubprogramName*, *constrainConsts*(*UnconstrainedConstList*)))

Figure E.19: constrain_consts critic

E.15.1 Behaviour

Where the goal contains unconstrained constants the viable_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique. The expectation is that an engineer will manually constrain the identified constants through the introduction of user rules. The engineer is responsible for ensuring the soundness of user rules. As SPADEase makes no soundness claims, it would be unsound for SPADEase to automate this task.

E.16 Critic: constrain_vars

The constrain_vars critic is shown in Figure E.20 and described below. This critic is associated with the viable_goal method, as discussed in §E.13.

Critic:
constrain_vars
Parent method:
viable_goal
Goal:
Goal
Successful method preconditions:
uncoupled_entry_vars(*Goal*, *UncoupledVarList*) *UncoupledVarList* = [] unconstrained_consts_vars(*Goal*, *UnconstrainedConstList*, *UnconstrainedVarList*) *UnconstrainedConstList* = []
Failed method precondition:
UnconstrainedVarList = []
Preconditions:
∅
Effects:
match_global_context(*sourceSubprogram*(*SubprogramName*)) write_line('Unconstrained variable(s) detected.') abort_plan(*abstractPredicate*(*SubprogramName*, *constrainVars*(*UnconstrainedVarList*)))

Figure E.20: constrain_vars critic

E.16.1 Behaviour

Where the goal contains unconstrained variables the viable_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique, ultimately triggering program analysis.

E.17 Critic: tightly_constrain_vars

The tightly_constrain_vars critic is shown in Figure E.21 and described below. This critic is associated with the viable_goal method, as discussed in §E.13.

Critic:
tightly_constrain_vars
Parent method:
viable_goal
Goal:
Goal
Successful method preconditions:
uncoupled_entry_vars(*Goal*, *UncoupledVarList*) *UncoupledVarList* = [] unconstrained_consts_vars(*Goal*, *UnconstrainedConstList*, *UnconstrainedVarList*) *UnconstrainedConstList* = [] *UnconstrainedVarList* = [] under_constrained_vars(*Goal*, *UnderConstrainedVarList*)
Failed method precondition:
UnderConstrainedVarList = []
Preconditions:
∅
Effects:
match_global_context(*sourceSubprogram*(*SubprogramName*)) write_line('Under constrained variable(s) detected.') abort_plan(*abstractPredicate*(*SubprogramName*, *tightlyConstrainVars*(*UnderConstrainedVarList*)))

Figure E.21: tightly_constrain_vars critic

E.17.1 Behaviour

Where under constrained variables are identified the viable_goal method will fail, leading to an invocation of this critic. The critic displays a message to describe the situation and aborts the plan with an appropriate failure critique, ultimately triggering program analysis.

E.18 Method: rtc_goal

The rtc_goal method is shown in Figure E.22 and described below.

Method:
rtc_goal
Tactic:
null_tactic
Goal:
Goal
Preconditions:
Goal is run-time check or between different invariants.
get_goal_category(*GoalCat*) select(*GoalCat*, [*rtc*, *binv*], _)
Effects:
∅
Subgoals:
[*Goal*]

Figure E.22: rtc_goal method

E.18.1 Behaviour

This method is successful where the goal corresponds to either a run-time check or a transition between different invariants.

E.19 Method: inv_goal

The inv_goal method is shown in Figure E.23 and described below.

Method:
inv_goal
Tactic:
null_tactic
Goal:
Goal
Preconditions:
Goal is returning to same invariant.
get_goal_category(*rinv*)
Effects:
∅
Subgoals:
[*Goal*]

Figure E.23: inv_goal method

E.19.1 Behaviour

This method is successful where the goal corresponds to a transition from an invariant returning to the same invariant.

E.20 Method: true_conc

The true_conc method is shown in Figure E.24 and described below.

Method:
true_conc
Tactic:
trivial_tactic
Goal:
$\_ : \_ \vdash \mathit{true}$
Preconditions:
$\emptyset$
Effects:
$\emptyset$
Subgoals:
[]

Figure E.24: true_conc method

E.20.1 Behaviour

This method identifies a goal that is immediately true. The trivial goal is discharged, leaving no subgoals.

E.21 Method: false_conc

The false_conc method is shown in Figure E.25 and described below.

Method:
false_conc
Tactic:
trivial_tactic
Goal:
LocalContextList : _ ⊢ *false*
Preconditions:
Retrieve recorded referenced variables.
get_constraining_vars(*LocalContextList*, *VarList*)
Check some variables have been referenced.
not(*VarList* = []),
Mark these variables as being potentially under constrained.
match_global_context(*sourceSubprogram*(*SubprogramName*))
add_under_constrained_vars(*SubprogramName*, *VarList*)
Never succeed.
fail
Effects:
∅
Subgoals:
[]

Figure E.25: false_conc method

E.21.1 Behaviour

This method identifies a goal that is immediately false. Such goals may arise on a branch of the proof tree, where the proof planner explores an unsuccessful chain of reasoning. Significantly, other unexplored branches may successfully lead to proof. To address this, in encountering a false goal, the plan is not aborted. Instead, those variables that contributed to the false goal are retrieved and recorded in the global context information associated with the plan. As described in §6.8.2, if the planning effort fails, a failure critique is raised, identifying all variables contributing to false goals as being potentially under constrained. This will ultimately trigger program analysis.

E.22 Method: linear_bounded_conc

The linear_bounded_conc method is shown in Figure E.26 and described below.

Method:
linear_bounded_conc
Tactic:
trivially_true_conc_tactic(*Conc*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Check that this is an inequality relation.
binary_explode(*Conc*, *Op*, _, _) is_inequality_op(*Op*)
Explore alternative conjunct forms.
sub_exp_polarity(*Conc*, *positive*, [], *Polarity*) select_alt_view_rule(*Polarity*, _, *true* : *Conc* ⇒ *ModifiedConc*)
Check the relation involves a simple linear expression and an integer.
binary_explode(*ModifiedConc*, _, *Left*, *Right*) simple_linear_exp_var(*Left*, *VarInt*) is_int(*Right*)
Find bounds for the integer variable in the simple linear expression.
int_bound_var(*HypList*, *VarInt*, *LowerInt*, *UpperInt*)
Effects:
Substitute and evaluate to find extreme points.
find_replace(*ModifiedConc*, *VarInt*, *LowerInt*, *Lowest*) find_replace(*ModifiedConc*, *VarInt*, *UpperInt*, *Highest*)
Determine if extreme points hold.
eval_exp((*Lowest* ∧ *Highest*), *ResultBool*)
Record the relevant variable.
add_constraining_vars(*LocalContextList*, [*VarInt*], *NewLocalContextList*)
Subgoals:
[*NewLocalContextList* : *HypList* ⊢ *ResultBool*]

Figure E.26: linear_bounded_conc method

E.22.1 Behaviour

This method applies a specific form of linear reasoning. The linear reasoning matches the automated reasoning capabilities of the Checker. This correspondence means that the tactic associated with the method simply instructs the Checker to automatically discharge the conclusion. While specific, the linear reasoning considered is frequently applicable where verifying exception freedom. In particular, the transitivity strategy may generate subgoals that are discharged by this method.

The method targets conclusions of the form:

$$linexp(VarInt)\ RelOp\ BoundInt \tag{E.3}$$

Where *linexp* is a simple linear expression parametrised by the integer variable *VarInt*,

RelOp is an inequality relation and *BoundInt* is an integer. The hypotheses are searched to discover tight constraints for *VarInt* such that:

$$(VarInt \geq LowerInt) \wedge (VarInt \leq UpperInt) \tag{E.4}$$

As *linexp* is linear, its extreme bounds coincide with the extreme values of *VarInt*. By substituting the bounds discovered for *VarInt*, the corresponding bounds of *linexp* can be determined. The conclusion is a relation comparing *linexp* with *BoundInt*. Thus the method reports the truth of the conclusion as the evaluation of the following expression:

$$\begin{aligned}&(linexp(LowerInt)\ RelOp\ BoundInt) \wedge \\ &(linexp(UpperInt)\ RelOp\ BoundInt)\end{aligned} \tag{E.5}$$

The truth of the conclusion depends on the quality of the constraints discovered for *VarInt*. This dependency is explicitly recorded in the local context information. Should the branch of reasoning fail, the false_conc method will suggest strengthening the constraints of *VarInt*.

E.23 Method: case_split

The case_split method is shown in Figure E.27 and described below.

Method:
case_split
Tactic:
case_split_tactic(*FirstCase*, *SecondCase*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Find multiplication of variable expressions.
exp_at(*Conc*, *Pos*, (*LeftVarInt* ∗ *RightVarInt*)) prog_var_exps(*LeftVarInt*, [*LeftVarInt*], _) prog_var_exps(*RightVarInt*, [*RightVarInt*], _)
Check neither parameter is exclusively negative or positive.
int_bound_var(*HypList*, *LeftVarInt*, *LeftLowerInt*, *LeftUpperInt*) (*LeftLowerInt* ∗ *LeftUpperInt*) < 0 int_bound_var(*HypList*, *RightVarInt*, *RightLowerInt*, *RightUpperInt*) (*RightLowerInt* ∗ *RightUpperInt*) < 0
Effects:
Establish case split for right variable.
FirstCase = (*RightVarInt* < 0) *SecondCase* = (*RightVarInt* ≥ 0)
Construct subgoals.
append(*HypList*, [*FirstCase*], *FirstCaseHypList*) append(*HypList*, [*SecondCase*], *SecondCaseHypList*)
Subgoals:
[*LocalContextList* : *FirstCaseHypList* ⊢ *Conc*, *LocalContextList* : *SecondCaseHypList* ⊢ *Conc*]

Figure E.27: case_split method

E.23.1 Behaviour

This method introduces a case split to ease the proof effort. Where reasoning about the multiplication of variables it is convenient if one of the variables is strictly negative or positive. For example, as part of the standard rules, the following rewrite rule is available:

$$((X * Z) \geq (Y * Z)) \Rightarrow (X \leq Y) \wedge (Z \leq 0) \tag{E.6}$$

Such a rewrite rule enables an inequality to be decomposed into separate conjuncts. However, this is only useful if it can be shown that Z is either zero or negative.

The method targets conclusions involving the multiplication of two variables:

$$LeftVarInt * RightVarInt \tag{E.7}$$

Further, the hypotheses must not constrain either of these variables to be strictly negative

or positive. In this situation, the following case split is introduced:

$$(RightVarInt < 0) \vee (RightVarInt \geq 0) \tag{E.8}$$

In each case, *RightVarInt* is either strictly negative or positive, easing the proof effort. Note that the variable on the right is targeted as several standard rules expect a strictly negative or positive variable on this side.

E.24 Method: mult_commute

The mult_commute method is shown in Figure E.28 and described below.

Method:
mult_commute
Tactic:
rewrite_tactic(*CommuteRewriteForm*, *conc*, *Conc*, *Pos*, *true* : (*LeftExp* ∗ *RightExp*) ⇒ (*RightExp* ∗ *LeftExp*))
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Find multiplication of expressions.
exp_at(*Conc*, *Pos*, (*LeftExp* ∗ *RightExp*))
Check left expression is integer and right expression is not.
is_int(*LeftExp*) not(is_int(*RightExp*))
Effects:
Find rule to commute multiplication of expressions.
sub_exp_polarity(*Conc*, *positive*, *Pos*, *Polarity*) select_alt_view_rule(*Polarity*, *CommuteRewriteForm*, *true* : (*LeftExp* ∗ *RightExp*) ⇒ (*RightExp* ∗ *LeftExp*))
Generate subgoal.
replace_at(*Conc*, *Pos*, (*RightExp* ∗ *LeftExp*), *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.28: mult_commute method

E.24.1 Behaviour

This method normalises the multiplication of an expression and an integer so that the integer appears on the right hand side. This supports the application of standard rules that expect an integer on the right hand side of a multiplication.

E.25 Method: eval_conc

The eval_conc method is shown in Figure E.29 and described below.

Method:
eval_conc
Tactic:
rewrite_tactic(*evaluate*(*SubExp*, *Value*), *conc*, *Conc*, *Pos*, *true* : *SubExp* ⇒ *Value*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Consider all conclusion subexpressions.
exp_at(*Conc*, *Pos*, *SubExp*)
Try to evaluate this subexpression.
eval_exp(*SubExp*, *Value*)
Check evaluated result is different.
not(*SubExp* = *Value*)
Succeed at most once.
cut
Effects:
Generate subgoal.
replace_at(*Conc*, *Pos*, *Value*, *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.29: eval_conc method

E.25.1 Behaviour

This method simplifies the conclusion by replacing an evaluable expression with the result of its evaluation. A cut is employed, preventing the exploration of alternative orderings of expression evaluations. The evaluation of both integer and boolean expressions is considered. An expression may be unchanged following its evaluation, for example the result of evaluating 10 remains 10. Thus, to ensure termination, the method is only successful where the evaluated expression is different to the result of its evaluation.

E.26 Method: split_conc_conj

The split_conc_conj method is shown in Figure E.30 and described below.

Method:
split_conc_conj
Tactic:
split_conc_conj_tactic(*LeftConc*, *RightConc*)
Goal:
LocalContextList : *HypList* ⊢ (*LeftConc* ∧ *RightConc*)
Preconditions:
∅
Effects:
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *LeftConc*, *LocalContextList* : *HypList* ⊢ *RightConc*]

Figure E.30: split_conc_conj method

E.26.1 Behaviour

This method simplifies a conjoined conclusion by introducing a separate subgoal for each conjunct.

E.27 Method: fertilize

The fertilize method is shown in Figure E.31 and described below.

Method:
fertilize
Tactic:
rewrite_tactic(*hypothesisFertilise*(*Conc*), *conc*, *Conc*, [], *true* : *Conc* ⇒ *true*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Ensure conclusion is not already true.
not(*Conc* = *true*)
Search for match with hypothesis.
select(*Conc*, *HypList*, _)
Effects:
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *true*]

Figure E.31: fertilize method

E.27.1 Behaviour

This method transforms a conclusion to *true* where it matches, or *fertilises*, against a hypothesis.

E.28 Method: clear_conc_exp

The clear_conc_exp method is shown in Figure E.32 and described below.

Method:
clear_conc_exp
Tactic:
rewrite_tactic(*ClearRewriteForm*, *conc*, *Conc*, *Pos*, *true* : *SubExp* ⇒ *NewSubExp*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Consider all expressions.
exp_at(*Conc*, *Pos*, *SubExp*)
Search to rewrite the expression.
sub_exp_polarity(*SubExp*, *positive*, *Pos*, *Polarity*) select_rewrite_rule(*Polarity*, *ClearRewriteForm*, *true* : *SubExp* ⇒ *NewSubExp*)
Check modified expression remains ground.
ground(*NewSubExp*)
Check modified expression is subexpression of original expression.
exp_at(*SubExp*, *SubPos*, *NewSubExp*) not(*SubPos* = [])
Succeed at most once.
cut
Effects:
Perform the rewrite.
replace_at(*Conc*, *Pos*, *NewSubExp*, *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.32: clear_conc_exp method

E.28.1 Behaviour

This method simplifies a conclusion by applying a rewrite rule that strictly eliminates expression structure. A cut is employed, preventing the exploration of alternative orderings of expression elimination. For example, a conclusion may be encountered of the following form:

$$(1 * x) = (0 * y) + z \tag{E.9}$$

The following two rewrite rules are available:

$$1 * A \Rightarrow A \tag{E.10}$$

$$0 * A \Rightarrow 0 \tag{E.11}$$

These rewrite rules eliminate expression structure, as the right hand side is a subset of the left hand side. Thus, successive applications of this method will simplify the above conclusion to:

$$x = z \tag{E.12}$$

E.29 Method: elim_var_conc

The elim_var_conc method is shown in Figure E.33 and described below.

Method:
elim_var_conc
Tactic:
Tactic
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Identify duplicate variables in conclusion.
prog_var_exps(*Conc*, *VarList*, _) filter_duplicates(*VarList*, *DuplVarList*, _)
Consider a duplicate variable.
select(*DuplVar*, *DuplVarList*, _)
Seek to cancel out the duplicate variables.
eliminate_duplicate_vars(2, *DuplVar*, *positive*, *Conc*, *Tactic*, *NewConc*)
Effects:
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.33: elim_var_conc method

E.29.1 Behaviour

In developing these proof plans, there was an occasional need for expression simplifications. Three alternative strategies were considered:

- **Measure reducing simplification** - A measure of expression simplicity is established, based on the number and type of arithmetic operators. The available rewrite rules are then explored, applying those that make the conclusion measurably simpler. The strategy is relatively simple, and effective in many cases. However, in

general, it is difficult to predict how the strategy will behave. The fundamental problem is that the strategy is not motivated by strong mathematical intuitions.

- **Annotation guided cancellation** - The cancellation of expressions is a mathematically intuitive simplification strategy. Similar to rippling, annotations may be introduced to guide the strategy. The conclusion may be annotated to identify pairs of expression structures that are candidates for cancellation. Further, rewrite rules may be annotated to identify those that move these expressions closer together. A search of the relevant rewrite rules is performed, until either the expressions are cancelled or the process terminates. The resulting strategy consistently performs valuable simplifications. However, sophisticated annotations would be required to discover all cancellations, particularly those that involve first moving expressions further apart. Also, an annotation guided strategy is a considerably complex mechanism for achieving a relatively trivial portion of mathematical reasoning.

- **Depth constrained cancellation** - Focusing on the cancellation of variables is a mathematically intuitive simplification strategy. Duplicate variables can be identified and manipulated without supporting annotations. Further, relevant rewrite rules can be identified as those manipulating the duplicate variables. To ensure termination, a depth constrained search of the relevant rewrite rules is performed. The resulting strategy is straightforward, and consistently performs valuable simplifications.

Depth constrained cancellation offers an effective balance between predictability, performance and reasoning capability. In particular, the elimination of duplicate variables can ease an application of the transitivity strategy.

E.30 Method: transitivity_entry

The transitivity_entry method is shown in Figure E.34 and described below.

Method:
transitivity_entry
Tactic:
sequence_tactic([*ModifyTactic*, *TransTactic*])
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Explore conclusion inequalities in both directions.
sub_exp_polarity(*Conc*, *positive*, [], *Polarity*) select_alt_view_rule(*Polarity*, *ModifyRewriteForm*, *true* : *Conc* ⇒ *ModifiedConc*) binary_explode(*ModifiedConc*, *Op*, *LeftExp*, *RightExp*) is_inequality_op(*Op*)
Check the left side contains the most program variables.
prog_var_exps(*LeftExp*, _, *LeftTotal*) prog_var_exps(*RightExp*, _, *RightTotal*) *LeftTotal* ≥ *RightTotal*
Check the left side has targeted operators.
total_functions(*LeftExp*, [+, −, ∗, *div*], *LeftCountInt*) *LeftCountInt* > 0
Search for applicable transitivity rewrite rule.
select_transitivity_rule(*Polarity*, *TransRewriteForm*, *true* : *ModifiedConc* ⇒ *NewConc*)
Effects:
ModifyTactic = rewrite_tactic(*ModifyRewriteForm*, *conc*, *Conc*, [], *true* : *Conc* ⇒ *ModifiedConc*) *TransTactic* = rewrite_tactic(*TransRewriteForm*, *conc*, *ModifiedConc*, [], *true* : *ModifiedConc* ⇒ *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.34: transitivity_entry method

E.30.1 Behaviour

This method introduces a transitivity step. The method is applicable where the conclusion is of the form:

$$Exp_1\ RelOp_1\ Exp_2 \tag{E.13}$$

Where Exp_i are expressions and $RelOp_j$ are inequality relations. Conclusions of this general form frequently occur where verifying exception freedom. Directly proving such conclusions can be difficult where the expressions involve numeric operations. However, the conclusion can often be made more tractable by decomposing the inequality into less

complex inequalities. The method achieves such a decomposition through the application of a transitive rewrite rule.

The method orients the conclusion inequality to target decomposition at its more complex side. The left hand side of the conclusion inequality must contain at least the same number of variables as the right hand side. Further, the left hand side must contain some numeric operations. A transitive rewrite rule is sought that transforms the conclusion as follows:

$$Exp_1\ RelOp_1\ Exp_2 \Rightarrow (Exp_1\ RelOp_1\ Z_1) \wedge (Z_1\ RelOp_2\ Exp_2) \tag{E.14}$$

The rewrite leads to the introduction of a meta-variable Z_1. The instantiation of this meta-variable requires a creative *eureka* step, as its form is not obvious from the surrounding structure. Using middle-out reasoning, the choice of the meta-variable is delayed, to be incrementally instantiated from future proof efforts.

For example, consider the SumArray subprogram shown in Figure E.35. This subprogram sums the values of an array. In verifying exception freedom, two goals are generated to verify that the assignment `R:=R+A(I)` always assigns a value to *r* that is within its lower and upper bound. The essential components of the upper bound goal, following the initialisation method, are shown below:

$$\begin{array}{l} (element(a,[i]) \leq 10) \wedge (r \leq i * 10) \wedge (i \leq 9) \\ \rightarrow \\ (r + element(a,[i])) \leq 100 \end{array} \tag{E.15}$$

Note that the conclusion matches the general conclusion of (E.13). Further, the conclusion contains a numeric operation that is hindering its immediate proof and there exists a transitive rewrite rule of the form:

$$A \leq B \Rightarrow (A \leq C) \wedge (C \leq B) \tag{E.16}$$

Together, these features support an application of this method, transforming the conclusion of (E.15) to:

$$(r + element(a,[i]) \leq C) \wedge (C \leq 100) \tag{E.17}$$

```
package SumArray_Package is
  subtype I_T is Integer range 0..9;
  subtype AE_T is Integer range 0..10;
  type A_T is array (I_T) of AE_T;
  subtype R_T is Integer range
    AE_T'First*((I_T'Last-I_T'First)+1)..
    AE_T'Last*((I_T'Last-I_T'First)+1);
  procedure SumArray(A: in  A_T;
                     R: out R_T);
  --# derives R from A;
end SumArray_Package;
```

```
package body SumArray_Package is
  procedure SumArray(A: in  A_T;
                     R: out R_T)
  is
  begin
    R:=0;
    for I in I_T loop
      --# assert R>=0 and R<=I*10;
      R:=R+A(I);
    end loop;
  end SumArray;
end SumArray_Package;
```

Figure E.35: SumArray subprogram

E.31 Method: transitivity_decomp

The transitivity_decomp method is shown in Figure E.36 and described below.

Method:
transitivity_decomp
Tactic:
rewrite_tactic(*DecompRewriteForm*, *conc*, *Conc*, *Pos*, *true* : *ConcConj* ⇒ *LeftConj* ∧ *RightConj*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Select a conjunct for decomposition, excluding the right most conjunct.
conjunct_at(*Conc*, *Pos*, *ConcConj*) not(*Pos* = [2])
Check the conjunct contains meta-variables.
not(ground(*ConcConj*))
Only decompose those conjuncts that contain targeted operators.
total_functions(*ConcConj*, [+, −, ∗, *div*], *BeforeCountInt*) *BeforeCountInt* > 0
Search to rewrite the conjunct.
sub_exp_polarity(*Conc*, *positive*, *Pos*, *Polarity*) select_rewrite_rule(*Polarity*, *DecompRewriteForm*, *true* : *ConcConj* ⇒ *LeftConj* ∧ *RightConj*)
Aim to exploit structure in hypothesis, so must not be ground.
not(ground(*LeftConj* ∧ *RightConj*)
Only accept decompositions that reduce targeted operators.
total_functions(*LeftConj* ∧ *RightConj*, [+, −, ∗, *div*], *AfterCountInt*) *BeforeCountInt* > *AfterCountInt*
Effects:
Perform the decomposition rewrite.
replace_at(*Conc*, *Pos*, *LeftConj* ∧ *RightConj*, *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.36: transitivity_decomp method

E.31.1 Behaviour

This method develops a transitivity step. Following an application of the transitivity_entry method, the conclusion will take the following general form:

$$(Exp_1\ RelOp_1\ Z_1) \wedge (Z_1\ RelOp_2\ Exp_2) \tag{E.18}$$

Where Exp_i are expressions, $RelOp_j$ are inequality relations and Z_k are meta-variables. This method decomposes inequality conjuncts. The method considers each conjunct, excluding the right most conjunct, that contains some numeric operations. A rewrite rule

is sought that performs a decomposition of the form:

$$(Exp_1\ RelOp_1\ Z_1) \Rightarrow (Exp_3\ RelOp_3\ Z_2) \wedge (Exp_4\ RelOp_4\ Z_3) \tag{E.19}$$

Which has the side effect of instantiating the meta-variable Z_1 as follows:

$$Z_1 = (Z_2\ F_l\ Z_3) \tag{E.20}$$

Where F_l are arithmetic functions. The number of numeric operations must decrease as a consequence of the decomposition. Note that, in some cases, meta-variables may be instantiated with expressions during decomposition.

Depending on the complexity of the initial conclusion, multiple invocations of decomposition may be required in completing the transitivity step. Following a complete decomposition, the conclusion will take the following general form:

$$\begin{aligned}&(Exp_n\ RelOp_n\ Z_n) \wedge \cdots \wedge (Exp_2\ RelOp_2\ Z_1)\\ &(Z_1\ F_1\ \ldots\ F_{n-1}\ Z_n)\ RelOp_1\ Exp_1\end{aligned} \tag{E.21}$$

For example, return to the SumArray subprogram introduced at the transitivity_entry method. Following an application of transitivity_entry the conclusion of the upper bound goal is transformed to:

$$(r + element(a, [i]) \leq C) \wedge (C \leq 100) \tag{E.22}$$

As the left most conjunct contains a numeric operator, it is a candidate for decomposition. A rewrite rule is available of the form:

$$((D + E) \leq (F + G)) \Rightarrow ((D \leq F) \wedge (E \leq G)) \tag{E.23}$$

The rewrite rule supports decomposing the left most conjunct, transforming the conclusion into:

$$((r \leq F) \wedge (element(a, [i]) \leq G)) \wedge (F + G \leq 100) \tag{E.24}$$

Note that, as a consequence of the rewrite, meta-variable C has been instantiated to $F + G$. Thus, through middle-out reasoning, the structure of C is emerging.

E.32 Method: transitivity_fertilize

The transitivity_fertilize method is shown in Figure E.37 and described below.

Method:
transitivity_fertilize
Tactic:
rewrite_tactic(*hypothesisFertilise*(*ConcConj*), *conc*, *Conc*, *Pos*, *true* : *ConcConj* ⇒ *true*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Select a conjunct for fertilisation, excluding the right most conjunct.
conjunct_at(*Conc*, *Pos*, *ConcConj*) not(*Pos* = [2])
Check the conjunct contains meta-variables.
not(ground(*ConcConj*))
Check for and retrieve single variable access.
prog_var_exps(*ConcConj*, [*VarAccess*], 1)
Search for match with hypothesis.
select(*ConcConj*, *HypList*, _)
Effects:
Record variable reference.
add_constraining_vars(*LocalContextList*, [*VarAccess*], *NewLocalContextList*)
Replace fertilised conjunct with true.
replace_at(*Conc*, *Pos*, *true*, *NewConc*)
Subgoals:
[*NewLocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.37: transitivity_fertilize method

E.32.1 Behaviour

This method continues the development of a transitivity step. The method instantiates meta-variables by matching inequality conjuncts against hypotheses. After potentially multiple applications of the transitivity_decomp method the conclusion will take the following general form:

$$\begin{aligned}&(Exp_n\ RelOp_n\ Z_n) \wedge \cdots \wedge (Exp_2\ RelOp_2\ Z_1)\\ &(Z_1\ F_1\ \ldots\ F_{n-1}\ Z_n)\ RelOp_1\ Exp_1\end{aligned} \tag{E.25}$$

Where Exp_i are expressions, $RelOp_i$ are inequality relations, Z_i are meta-variables and F_i are arithmetic functions. The method considers each conjunct, excluding the right most conjunct, that contains a single variable. A hypothesis is sought that constrains the bounds of this variable. Where available, the conjunct is *fertilised* against the hypothesis and is trivially eliminated. Note that, through backtracking, all applicable hypotheses will be considered. As a consequence of fertilisation, the meta-variables in the conjunct

will be instantiated with the constraints on the variable. Significantly, the proof of the remaining conclusion now depends on the quality of these constraints. The dependency is recorded by storing the name of the variable that contributed these constraints in the local context information. Should the overall proof fail, the false_conc method will suggest strengthening the constraints on these referenced variables.

For example, return to the SumArray subprogram introduced at the transitivity_entry method. Following an application of transitivity_decomp the essential upper bound goal will take the following form:

$$\begin{aligned} &(\mathit{element}(a,[i]) \leq 10) \wedge (r \leq i * 10) \wedge (i \leq 9) \\ &\rightarrow \\ &((r \leq F) \wedge (\mathit{element}(a,[i]) \leq G)) \wedge (F + G \leq 100) \end{aligned} \tag{E.26}$$

Note that the first and second conjuncts may be fertilised against hypotheses. Following two applications of the transitivity_fertilize method, the conclusion of the goal is transformed into the following:

$$(\mathit{true} \wedge \mathit{true}) \wedge ((i * 10) + 10 \leq 100) \tag{E.27}$$

E.33 Method: transitivity_close

The transitivity_close method is shown in Figure E.38 and described below.

Method:
transitivity_close
Tactic:
null_tactic
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Check the conclusion contains no meta-variables.
ground(*Conc*))
Effects:
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *Conc*]

Figure E.38: transitivity_close method

E.33.1 Behaviour

This method closes a transitivity step. Once every meta-variable has become instantiated then middle-out reasoning is complete and the transitivity step is successful.

For example, return to the SumArray subprogram introduced at the transitivity_entry method. Following applications of transitivity_fertilize the essential upper bound goal will take the following form:

$$\begin{array}{l} (\mathit{element}(a, [i]) \leq 10) \wedge (r \leq i * 10) \wedge (i \leq 9) \\ \rightarrow \\ (\mathit{true} \wedge \mathit{true}) \wedge ((i * 10) + 10 \leq 100) \end{array} \tag{E.28}$$

The conclusion contains zero meta-variables, thus the transitivity step is complete. Note that the *proof residue* in the third conjunct will be proved via the linear_bounded_conc method.

E.34 Method: transitivity_unblock

The transitivity_unblock method is shown in Figure E.39 and described below.

Method:
transitivity_unblock
Tactic:
rewrite_tactic(*UnblockRewriteForm*, *conc*, *Conc*, *Pos*, *true* : *ConcConj* ⇒ *true*)
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Select a conjunct, excluding the right most conjunct.
conjunct_at(*Conc*, *Pos*, *ConcConj*)
not(*Pos* = [2])
Check this conjunct has a meta-variable.
not(ground(*ConcConj*))
Check for zero variable accesses.
prog_var_exps(*ConcConj*, [], 0)
Check for zero numeric operations.
total_functions(*LeftExp*, [+, −, ∗, *div*], 0)
Search for rewrite rule that eliminates and grounds the conjunct.
sub_exp_polarity(*Conc*, *positive*, *Pos*, *Polarity*)
select_rewrite_rule(*Polarity*, *UnblockRewriteForm*, *true* : *ConcConj* ⇒ *true*)
ground(*ConcConj*)
Effects:
Replace the conjunct with true.
replace_at(*Conc*, *Pos*, *true*, *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.39: transitivity_unblock method

E.34.1 Behaviour

This method supports the development of a transitivity step. During the transitivity step, conjuncts may be encountered of the general form:

$$(Const\ RelOp\ Z) \tag{E.29}$$

Where *Const* is an constant, *RelOp* is an inequality relation and Z is a meta-variable. Neither transitivity_decomp nor transitivity_fertilize applies, thus the transitivity step is blocked. This method allows the transitivity step to continue by applying rewrite rules that trivially discharge such conjuncts. For example, where blocked by the conjunct:

$$(255 \leq A) \tag{E.30}$$

The following rewrite rule might be applied:

$$(A \leq A) \Rightarrow true \tag{E.31}$$

Supporting the elimination of the conjunct, and instantiating the meta-variable A as 255.

E.35 Method: ripple_entry

The ripple_entry method is shown in Figure E.40 and described below.

Method:
ripple_entry
Tactic:
null_tactic
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Search for induction hypothesis.
select(*Hyp*, *HypList*, _)
Attempt to annotate conclusion with respect to hypothesis.
ripple_annotate(*Hyp*, *Conc*, *AnnConc*)
Effects:
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *AnnConc*]

Figure E.40: ripple_entry method

E.35.1 Behaviour

This method introduces a ripple step into the proof. Rippling is supported through expression annotations. Where applicable, the method annotates the conclusion to begin an application of rippling. As the method does not modify the object-level goal, it is associated with the null_tactic.

As detailed in [BBHI05], rippling may employ several annotations to cater for different proof strategies. In our proof plans, only a core subset of annotations are employed. Nevertheless, these annotations are often sufficient to prove the relatively simple loop invariants that occur in verifying exception freedom.

Terms are annotated by placing markers around subterms. A *wave-front* is used to identify the boundaries of a particular subterm. The wave-front is illustrated by placing the subterm in a shaded box:

$$f(t_1, \ldots, t_n) \tag{E.32}$$

Every wave-front must contain at least one *wave-hole*. Each wave-hole identifies the

boundaries of a subterm that lies inside the wave-front. Each wave-hole is indicated by unshading the subterm within the wave-front:

$$f(\,t_1\,,\ldots,t_n) \tag{E.33}$$

The shaded syntax is described as being *in the wave-front* while the unshaded syntax is described as being *in the wave-hole*. An expression may be annotated with a number of wave-fronts:

$$f_1(\,t_1^1\,,\ldots,t_n^1) \wedge \cdots \wedge f_i(\,t_1^i\,,\ldots,t_n^i) \tag{E.34}$$

Where an expression is annotated, the *skeleton* refers to the expression that remains when removing all subexpressions that are in the wave-front:

$$t_1^1 \wedge \cdots \wedge t_1^i \tag{E.35}$$

The rippling annotations are used to describe the differences between an induction conclusion and its corresponding induction hypothesis. Consider the induction step case below:

$$f(i) \rightarrow f(s(i)) \tag{E.36}$$

The induction conclusion may be annotated so that its skeleton matches the induction hypothesis:

$$f(i) \rightarrow f(\,s(\,i\,)\,) \tag{E.37}$$

The annotations reveal that proof may be completed by eliminating $s(\ldots)$, the differing syntax in the wave-front.

For example, consider the SumMultTwinArray subprogram shown in Figure E.41. This subprogram sums the multiplication of elements at the same index in two arrays. Note that an invariant has been introduced to support the verification of exception freedom. The essential components of the lower bound invariant goal, following the initialisation method, are shown below:

$$\begin{aligned}
&(element(a1,[i]) \geq -10) \wedge (element(a1,[i]) \leq 10) \wedge \\
&(element(a2,[i]) \geq -10) \wedge (element(a2,[i]) \leq 10) \wedge \\
&(r \geq i * (-100)) \\
&\rightarrow \\
&r + (element(a1,[i]) * element(a2,[i])) \geq (i+1) * (-100)
\end{aligned} \tag{E.38}$$

The induction conclusion may be annotated against the induction hypothesis as follows,

supporting an application of rippling:

$$r + element(a1,[i]) * element(a2,[i]) \geq (i + 1) * (-100) \qquad \text{(E.39)}$$

```
package SumMultTwinArray_Package is
  subtype I_T is Integer range 0 .. 5;
  subtype AE_T is Integer range -10 .. 10;
  type A_T is array (I_T) of AE_T;
  subtype R_T is Integer range
    (AE_T'First*AE_T'Last)*((I_T'Last-I_T'First)+1)..
    (AE_T'Last*AE_T'Last)*((I_T'Last-I_T'First)+1);
  procedure SumMultTwinArray(A1: in A_T; A2: in A_T; R: out R_T);
  --# derives R from A1, A2;
end SumMultTwinArray_Package;
```

```
package body SumMultTwinArray_Package is
  procedure SumMultTwinArray(A1: in A_T; A2: in A_T; R: out R_T)
  is
  begin
    R:=0;
    for I in I_T loop
      --# assert R>=I*(-100) and R<=I*100;
      R:=R+(A1(I)*A2(I));
    end loop;
  end SumMultTwinArray;
end SumMultTwinArray_Package;
```

Figure E.41: SumMultTwinArray subprogram

E.36 Method: ripple_wave

The ripple_wave method is shown in Figure E.42 and described below.

Method:
ripple_wave
Tactic:
rewrite_tactic(*WaveRewriteForm*, *conc*, *EraseConc*, *Pos*, *true* : *EraseSubExp* ⇒ *EraseNewSubExp*)
Goal:
LocalContextList : *HypList* ⊢ *AnnConc*
Preconditions:
Consider all well annotated subterms in the conclusion.
ripple_exp_at(*AnnConc*, *Pos*, *AnnSubExp*)
Apply wave rule to the subterm.
ripple_erasure(*AnnConc*, *EraseConc*)
sub_exp_polarity(*EraseConc*, *positive*, *Pos*, *Polarity*)
select_wave_rule(*Polarity*, *WaveRewriteForm*, *true* : *AnnSubExp* ⇒ *NewAnnSubExp*)
Effects:
Perform the rewrite.
replace_at(*AnnConc*, *Pos*, *NewAnnSubExp*, *NewAnnConc*)
Generate erased forms for proof checking.
ripple_erasure(*AnnSubExp*, *EraseSubExp*)
ripple_erasure(*NewAnnSubExp*, *EraseNewSubExp*)
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewAnnConc*]

Figure E.42: ripple_wave method

E.36.1 Behaviour

This method continues the development of a ripple step. Following an application of the ripple_entry method the induction conclusion will be annotated, identifying its differences against an induction hypothesis. This method rewrites the conclusion such that these differences are moved outwards.

The available rewrite rules are restricted to *wave-rules*. A wave-rule is an annotated rewrite rule, which may only be applied where both the expression structure and annotations match. The wave-rules are automatically generated from the available rewrite rules. Significantly, the annotations are configured so that wave-rules simultaneously preserve similarities and *ripples* the differences outwards. The application of wave-rules involves a tightly constrained search and is guaranteed to terminate. For illustration consider the rewrite rule:

$$f(s(X)) \Rightarrow f(X) \wedge g(X) \tag{E.40}$$

Which may be annotated as the wave-rule:

$$f(\ s(\ X\ )\ ) \Rightarrow\ f(X)\ \wedge g(X) \tag{E.41}$$

This wave-rule preserves similarities as the skeleton $f(X)$, yet moves differences outwards as $g(X)$. Consider the annotated induction goal below:

$$f(i) \rightarrow f(\ s(\ i\ )\ ) \tag{E.42}$$

The wave-rule (E.41) matches with the induction conclusion rewriting the goal to:

$$f(i) \rightarrow\ f(i)\ \wedge g(i) \tag{E.43}$$

The differences are now fully ripped outwards, with the induction conclusion containing an embedding of the induction hypothesis. In general, multiple wave-rule rewrites may be required to ripple all differences outwards.

For example, return to the SumMultTwinArray subprogram introduced at the ripple_entry method. Following an application of ripple_entry the induction conclusion of the lower bound invariant goal is annotated as follows:

$$r\ + element(a1,[i]) * element(a2,[i])\ \geq (\ i\ + 1\ ) * (-100) \tag{E.44}$$

The following two wave-rules are available:

$$(\ A\ + B\ ) * C \Rightarrow\ (A * C)\ + (B * C) \tag{E.45}$$

$$(\ A\ + C)\ \geq\ (\ B\ + D)\ \Rightarrow\ (A \geq B)\ \wedge (C \geq D) \tag{E.46}$$

Applying wave-rule (E.45) to the right hand side of the induction conclusion gives:

$$r\ + element(a1,[i]) * element(a2,[i])\ \geq (\ (i * (-100))\ + (1 * (-100))\ ) \tag{E.47}$$

Applying (E.46) to the transformed conclusion gives:

$$r \geq (i * (-100))\ \wedge element(a1,[i]) * element(a2,[i]) \geq (1 * (-100)) \tag{E.48}$$

Resulting in the differences being fully ripped outwards.

E.37 Method: ripple_fertilize

The ripple_fertilize method is shown in Figure E.43 and described below.

Method:
ripple_fertilize
Tactic:
rewrite_tactic(*hypothesisFertilise*(*IndHyp*), *conc*, *Conc*, *IndHypPos*, *true* : *IndHyp* ⇒ *true*)
Goal:
LocalContextList : *HypList* ⊢ *AnnConc*
Preconditions:
Check the conclusion is rippled fully outwards.
ripple_complete(*AnnConc*, *IndHyp*, *Conc*, *IndHypPos*)
Effects:
Replace fertilised induction hypothesis with true.
replace_at(*Conc*, *IndHypPos*, *true*, *NewConc*)
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *NewConc*]

Figure E.43: ripple_fertilize method

E.37.1 Behaviour

This method completes a ripple step. Following applications of the ripple_wave method, the induction conclusion may become fully rippled. This method eliminates the embedded induction hypothesis, leaving the rippled out differences as a proof residue.

For illustration consider the fully rippled induction goal below:

$$f(i) \rightarrow \boxed{f(i)} \wedge g(i) \tag{E.49}$$

The embedded induction hypothesis in the induction conclusion may be *fertilised* against the actual induction hypothesis and trivially eliminated. As this completes the ripple step, all remaining annotations are cleared. Following fertilisation the goal above becomes:

$$f(i) \rightarrow true \wedge g(i) \tag{E.50}$$

For example, return to the SumMultTwinArray subprogram introduced at the ripple_entry method. Following applications of ripple_wave the essential components of

the lower bound invariant goal are as follows:

$$\begin{array}{l}(element(a1,[i]) \geq -10) \wedge (element(a1,[i]) \leq 10) \wedge \\ (element(a2,[i]) \geq -10) \wedge (element(a2,[i]) \leq 10) \wedge \\ (r \geq i * (-100)) \\ \rightarrow \\ r \geq (i * (-100)) \wedge element(a1,[i]) * element(a2,[i]) \geq (1 * (-100))\end{array} \tag{E.51}$$

The induction conclusion contains an embedding of the induction hypothesis. Fertilisation may be performed, completing the ripple step and rewriting the conclusion as follows:

$$true \wedge element(a1,[i]) * element(a2,[i]) \geq (1 * (-100)) \tag{E.52}$$

The proof residue is discharged through the run_time_check strategy.

E.38 Method: ripple_unblock

The ripple_unblock method is shown in Figure E.44 and described below.

Method:
ripple_unblock
Tactic:
Tactic
Goal:
LocalContextList : *HypList* ⊢ *AnnConc*
Preconditions:
Explore unblocking strategies.
ripple_unblock_strategies(*AnnConc*, *UnblockedAnnConc*, *Tactic*)
Effects:
∅
Subgoals:
[*LocalContextList* : *HypList* ⊢ *UnblockedAnnConc*]

Figure E.44: ripple_unblock method

E.38.1 Behaviour

This method supports the development of a ripple step. Rippling becomes blocked where no wave-rules are applicable to the annotated conclusion. However, internally transforming the conclusion or its annotations may enable the ripple step to continue. The following two unblocking strategies are attempted:

- **Simplify annotations** - The simplification of annotations can increase the applicability of wave-rules. In particular a wave-front may be nested entirely inside a

wave-hole as show below:

$$f(\boxed{t_1}, t_2, \ldots, t_n) \tag{E.53}$$

Such annotations are simplified by removing the nested wave-front:

$$f(\boxed{t_1}, t_2, \ldots, t_n) \tag{E.54}$$

- **Move wave-hole outwards** - Moving the location of a wave-hole outwards can increase the applicability of wave-rules. For illustration, consider the annotated expression below:

$$(\boxed{t_1} + t_2) + t_3 \tag{E.55}$$

To manipulate this expression, the left hand side of wave-rules must take the following form:

$$(\boxed{A} + B) + C \tag{E.56}$$

By exploiting commutativity of plus, the wave-hole of (E.55) may be moved outwards as follows:

$$\boxed{A} + (B + C) \tag{E.57}$$

Significantly, to manipulate the transformed expression, the left hand side of wave-rules may now take the more general form:

$$\boxed{A} + B \tag{E.58}$$

E.39 Proof Plans for Program Analysis Queries

Proof plans are developed to answer program analysis queries, as detailed in the following sections. As described in Chapter 5, the program analyser does not verify the correctness of its discovered invariants. Thus, proof plans specific to program analysis are not checked at the object-level, and are not associated with corresponding tactics.

E.40 Strategy: pa_exp_simplify

The pa_exp_simplify strategy is shown in Figure E.45 and described below.

Waterfall:	
pa_exp_simplify	
Actions:	
prune_conc_duplicate	↦ pa_exp_simplify
prune_conc_eq	↦ pa_exp_simplify
eval_conc	↦ pa_exp_simplify
clear_conc_exp	↦ pa_exp_simplify
report_conc	↦ ∅

Figure E.45: pa_exp_simplify strategy

E.40.1 Behaviour

This strategy simplifies expressions. The strategy expects contextual information to be presented as hypotheses and the target expression to be presented as a conclusion.

E.41 Strategy: pa_exp_constrain

The pa_exp_constrain strategy is shown in Figure E.46 and described below.

Waterfall:
pa_exp_constrain
Actions:
specialise_hyps ↦ pa_exp_constrain1

Waterfall:
pa_exp_constrain1
Actions:
solve_eq_hyp_for_var ↦ pa_exp_constrain1
constrain_conc_conj ↦ ∅

Figure E.46: pa_exp_constrain strategy

E.41.1 Behaviour

This strategy discovers bounds for numeric expressions. The strategy expects contextual information to be presented as hypotheses and the target numeric expression to be presented as a conclusion.

E.42 Strategy: pa_spark_exp

The pa_spark_exp strategy is shown in Figure E.47 and described below.

Waterfall:
pa_spark_exp
Actions:
specialise_hyps ↦ pa_spark_exp1

Waterfall:	
pa_spark_exp1	
Actions:	
prune_conc_duplicate	↦ pa_spark_exp1
prune_conc_eq	↦ pa_spark_exp1
eval_conc	↦ pa_spark_exp1
clear_conc_exp	↦ pa_spark_exp1
solve_eq_hyp_for_var	↦ pa_spark_exp1
elim_aux_var_via_eq	↦ pa_spark_exp1
elim_prog_var_exp_via_eq	↦ pa_spark_exp1
elim_aux_var_via_int_arith	↦ pa_spark_exp1
is_spark_exp	↦ ∅

Figure E.47: pa_spark_exp strategy

E.42.1 Behaviour

This strategy transforms an expression into a form which can be directly expressed in SPARK annotations. The strategy expects contextual information to be presented as hypotheses and the target expression to be presented as a conclusion.

E.43 Strategy: pa_disj_norm_form

The pa_disj_norm_form strategy is shown in Figure E.48 and described below.

Waterfall:
pa_disj_norm_form
Actions:
disj_norm_form ↦ pa_disj_norm_form
report_conc ↦ ∅

Figure E.48: pa_disj_norm_form strategy

E.43.1 Behaviour

This strategy transforms an expression into disjunctive normal form. The strategy ignores any hypotheses and expects the target expression to be presented as a conclusion.

E.44 Method: prune_conc_duplicate

The prune_conc_duplicate method is shown in Figure E.49 and described below.

Method:
prune_conc_duplicate
Tactic:
$\emptyset$
Goal:
LocalContextList : *HypList* $\vdash$ *Conc*
Preconditions:
Remove duplicate conjuncts.
exp_explode(*Conc*, $\wedge$, *ConcList*) filter_duplicates(*ConcList*, _, *NoDupConjList*) exp_explode(*NoDupConj*, $\wedge$, *NoDupConjList*)
Remove duplicate disjuncts.
exp_explode(*NoDupConj*, $\vee$, *NoDupConjList*) filter_duplicates(*NoDupConjList*, _, *NoDupConjDisjList*) exp_explode(*NoDupConjDisj*, $\vee$, *NoDupConjDisjList*)
Check that some duplicates were removed.
not(*NoDupConjDisj* = *Conc*)
Effects:
$\emptyset$
Subgoals:
[*LocalContextList* : *HypList* $\vdash$ *NoDupConjDisj*]

Figure E.49: prune_conc_duplicate method

E.44.1 Behaviour

The method removes all duplicate conjuncts and disjuncts from a conclusion. Such duplication occurs frequently during program analysis, as properties are combined at program merge points.

E.45 Method: prune_conc_eq

The prune_conc_eq method is shown in Figure E.50 and described below.

Method:
prune_conc_eq
Tactic:
$\emptyset$
Goal:
LocalContextList : *HypList* $\vdash$ *Conc*
Preconditions:
Search for equality conjuncts in both directions.
conjunct_at(*Conc*, *Pos*, *ConcConj*) sub_exp_polarity(*Conc*, *positive*, *Pos*, *Polarity*) select_alt_view_rule(*Polarity*, _, *true* : *ConcConj* $\Rightarrow$ (*LeftExp* = *RightExp*))
Check is unconstrained variable equality.
unconstrained_var(*LeftExp*)
Succeed at most once.
cut
Effects:
Replace equality conjunct with true.
replace_at(*Conc*, *Pos*, *true*, *InterConc*)
Replace the unconstrained variable with its expression.
find_replace(*InterConc*, *LeftExp*, *RightExp*, *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* $\vdash$ *NewConc*]

Figure E.50: prune_conc_eq method

E.45.1 Behaviour

This method identifies conjuncts that relate an unconstrained variable to an expression. Such conjuncts occur frequently during program analysis, as unconstrained variables are introduced to ease the transformation of expressions. Where found, the conjunct is replaced with true, and all occurrences of the unconstrained variable are replaced with its corresponding expression.

E.46 Method: report_conc

The report_conc method is shown in Figure E.51 and described below.

Method:
report_conc
Tactic:
Conc
Goal:
_ : _ ⊢ *Conc*
Preconditions:
∅
Effects:
∅
Subgoals:
[]

Figure E.51: report_conc method

E.46.1 Behaviour

This method is always successful, returning the conclusion of the goal through the tactic slot. The method leaves no subgoals.

E.47 Method: solve_eq_hyp_for_var

The solve_eq_hyp_for_var method is shown in Figure E.52 and described below.

Method:
solve_eq_hyp_for_var
Tactic:
∅
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Consider each equality hypothesis.
select(*Hyp*, *HypList*, _) select(*Hyp* = (_ = _)
Solve for variables referenced in the hypothesis.
prog_var_exps(*Hyp*, *ProgVarExpList*, _) aux_vars(*Hyp*, *AuxVarList*, _) append(*ProgVarExpList*, *AuxVarList*, *VarList*) select(*Var*, *VarList*, _) solve_for_var(*Hyp*, *Var*, *SolvedEq*)
Check the solved equality is not already present.
not(select(*SolvedEq*, *HypList*, _))
Succeed at most once.
cut
Effects:
∅
Subgoals:
[*LocalContextList* : [*SolvedEq*|*HypList*] ⊢ *Conc*]

Figure E.52: solve_eq_hyp_for_var method

E.47.1 Behaviour

This method introduces additional hypotheses by solving existing equality hypotheses for their referenced variables. The method exploits the computer algebra system YACAS [YAC] to perform the equation solving. In particular, the default capabilities of the *Solve* function are used[2]. For example, a hypothesis of the following form may be encountered:

$$a = (b + c) + d \tag{E.59}$$

To solve for c the following query may be sent to YACAS:

$$Solve(a = (b + c) + d, c) \tag{E.60}$$

[2]In practice, to minimise implementation effort, this method does not communicate directly with YACAS. Instead, a look-up table is maintained, describing the capabilities of YACAS for each equality encountered.

YACAS is successful, presenting the result:

$$c = a - (b + d) \tag{E.61}$$

Note that it would be difficult to generate a tactic that describes the actions of YACAS at the object-level. However, as this method is only applied during program analysis, such a tactic is not required.

E.48 Method: constrain_conc_conj

The constrain_conc_conj method is shown in Figure E.53 and described below.

Method:
constrain_conc_conj
Tactic:
$(Conc \geq LowerInt) \wedge (Conc \leq UpperInt)$
Goal:
$LocalContextList : HypList \vdash Conc$
Preconditions:
Find bounds for the integer exp conclusion.
int_bound_var($HypList$, $Conc$, $LowerInt$, $UpperInt$)
Effects:
$\emptyset$
Subgoals:
[]

Figure E.53: constrain_conc_conj method

E.48.1 Behaviour

This method discovers bounds for the conclusion expression, returning these through the tactic slot. The method leaves no subgoals.

E.49 Method: elim_aux_var_via_eq

The elim_aux_var_via_eq method is shown in Figure E.54 and described below.

Method:
elim_aux_var_via_eq
Tactic:
∅
Goal:
LocalContextList : *HypList* ⊢ *Conc*
Preconditions:
Consider each equality hypothesis.
select(*Hyp*, *HypList*, *RemHypList*) *Hyp* = (_ = _)
Explore equality hypothesis in both directions.
sub_exp_polarity(*Hyp*, *negative*, [], *Polarity*) select_alt_view_rule(*Polarity*, _, *true* : *Hyp* ⇒ (*LeftExp* = *RightExp*))
Check that the left expression is an auxiliary variable.
aux_vars(*LeftExp*, [*LeftExp*], 1)
Check that the right expression contains no auxiliary variables.
aux_vars(*RightExp*, [], 0)
Succeed at most once.
cut
Effects:
Remove equality hypothesis.
find_replace((*RemHypList*, *Conc*), *LeftExp*, *RightExp*, (*NewHypList*, *NewConc*))
Subgoals:
[*LocalContextList* : *NewHypList* ⊢ *NewConc*]

Figure E.54: elim_aux_var_via_eq method

E.49.1 Behaviour

This method eliminates auxiliary variables in the goal. The method identifies a hypothesis equality between an auxiliary variable and an expression that has no auxiliary variables. Where found, the hypothesis equality is eliminated and all occurrences of the auxiliary variable are replaced with its equivalent expression.

E.50 Method: elim_prog_var_exp_via_eq

The elim_prog_var_exp_via_eq method is shown in Figure E.55 and described below.

Method:
elim_prog_var_exp_via_eq
Tactic:
$\emptyset$
Goal:
LocalContextList : *HypList* $\vdash$ *Conc*
Preconditions:
Consider each equality hypothesis.
select(*Hyp*, *HypList*, *RemHypList*) *Hyp* = (_ = _)
Explore equality hypothesis in both directions.
sub_exp_polarity(*Hyp*, *negative*, [], *Polarity*) select_alt_view_rule(*Polarity*, _, *true* : *Hyp* $\Rightarrow$ (*LeftExp* = *RightExp*))
Check that the left expression is a program variable.
prog_var_exps(*LeftExp*, [*LeftExp*], 1)
Check that the right expression contains no variables.
prog_var_exps(*RightExp*, [], 0) aux_vars(*RightExp*, [], 0)
Succeed at most once.
cut
Effects:
Eliminate program variable expression.
find_replace((*RemHypList*, *Conc*), *LeftExp*, *RightExp*, (*NewHypList*, *NewConc*))
Subgoals:
[*LocalContextList* : *NewHypList* $\vdash$ *NewConc*]

Figure E.55: elim_prog_var_exp_via_eq method

E.50.1 Behaviour

This method simplifies the goal by eliminating program variables. The method identifies a hypothesis equality between a program variable and an expression that has no variables. Where found, the hypothesis equality is eliminated and all occurrences of the program variable are replaced with its equivalent expression.

E.51 Method: elim_aux_var_via_int_arith

The elim_aux_var_via_int_arith method is shown in Figure E.56 and described below.

Method:
elim_aux_var_via_int_arith
Tactic:
$\emptyset$
Goal:
LocalContextList : *HypList* $\vdash$ *Conc*
Preconditions:
Select a conclusion conjunct.
conjunct_at(*Conc*, *Pos*, *ConcConj*)
Select an auxiliary variables in the conjunct.
aux_vars(*ConcConj*, *AuxVarList*, _) select(*AuxVar*, *AuxVarList*, _)
Eliminate the auxiliary variable through interval reasoning.
elim_bounded_var(*HypList*, *AuxVar*, *ConcConj*, *NewConcConj*)
Succeed at most once.
cut
Effects:
Adopt conjunct with auxiliary variable eliminated.
replace_at(*Conc*, *Pos*, *NewConcConj*, *NewConc*)
Subgoals:
[*LocalContextList* : *NewHypList* $\vdash$ *NewConc*]

Figure E.56: elim_aux_var_via_int_arith method

E.51.1 Behaviour

This method eliminates auxiliary variables in the goal. The method applies a restricted form of interval reasoning to replace the auxiliary variable with its known bounds.

E.52 Method: is_spark_exp

The is_spark_exp method is shown in Figure E.57 and described below.

Method:
is_spark_exp
Tactic:
Conc
Goal:
$\_ : \_ \vdash Conc$
Preconditions:
Check that the conclusion contains no auxiliary variables.
aux_vars(*Conc*, [], 0)
Effects:
$\emptyset$
Subgoals:
[]

Figure E.57: is_spark_exp method

E.52.1 Behaviour

This method is successful where the conclusion can be directly expressed in SPARK annotations. In this case, the conclusion is returned through the tactic slot. The method leaves no subgoals.

E.53 Method: disj_norm_form

The disj_norm_form method is shown in Figure E.58 and described below.

Method:
disj_norm_form
Tactic:
$\emptyset$
Goal:
LocalContextList : *HypList* $\vdash$ *Conc*
Preconditions:
Consider all expressions.
exp_at(*Conc*, *Pos*, *SubExp*)
Find rewrite that moves toward disjunctive normal form.
sub_exp_polarity(*Conc*, *positive*, *Pos*, *Polarity*) select_rewrite_rule(*Polarity*, *RewriteForm*, *true* : *SubExp* $\Rightarrow$ *NewSubExp*) *RewriteForm* = *rule*(_, *rlu*, *dnf*(_), *normal*)
Succeed at most once.
cut
Effects:
Perform disjunctive normal form rewrite.
replace_at(*Conc*, *Pos*, *NewSubExp*, *NewConc*)
Subgoals:
[*LocalContextList* : *HypList* $\vdash$ *NewConc*]

Figure E.58: disj_norm_form method

E.53.1 Behaviour

This method is successful where the conclusion can be brought closer to disjunctive normal form. The method applies rewrite rules that move an expression toward disjunctive normal form, as detailed in §B.4.4.

Appendix F

MiniSPARK Grammar

F.1 Introduction

As discussed in §7.2, program analysis is performed on a subset of SPARK as MiniSPARK. The complete grammar of MiniSPARK is listed below. Note that the tokenizer suppresses the content of SPARK annotations, leading to a smaller grammar.

F.2 Grammar

<CompilationUnit> ::= *<PackageDeclaration>* |
<PackageDeclaration> *<PackageDeclaration>* |
<PackageBody>

<PackageDeclaration> ::= *rwpackage* *<DottedSimpleName>*
rwis *<VisiblePartRep>*
rwend *<DottedSimpleName>*
semicolon

<VisiblePartRep> ::= *<VisiblePartRep>* *<RestrictedBasicDeclaration>* |
<VisiblePartRep> *<SubprogramDeclaration>* |
null

<RestrictedBasicDeclaration> ::= *<ConstantDeclaration>* |
<SubtypeDeclaration> |
<FullTypeDeclaration>

<ConstantDeclaration> ::= *Identifier* *colon* *<Rwconstant>*
becomes *<Expression>*
semicolon

<SubtypeDeclaration> ::= *rwsubtype* *Identifier*
rwis *<TypeMark>*
rwrange *<Arange>*
semicolon

<FullTypeDeclaration> ::= *rwtype* *Identifier*
rwis *<TypeDefinition>*
semicolon

<TypeDefinition> ::= <ConstrainedArrayDefinition> |
<IntegerTypeDefinition>

<ConstrainedArrayDefinition> ::= rwarray leftparen
<TypeMark>
rightparen rwof <TypeMark>

<IntegerTypeDefinition> ::= <RangeConstraint>

<RangeConstraint> ::= rwrange <SimpleExpression>
doubledot <SimpleExpression>

<Arange> ::= <SimpleExpression> doubledot <SimpleExpression>

<SubprogramDeclaration> ::= <ProcedureSpecification>
semicolon <ProcedureAnnotation> |
<FunctionSpecification>
semicolon <FunctionAnnotation>

<ProcedureAnnotation> ::= <ProcedureConstraint> |
<DependencyRelation> <ProcedureConstraint>

<FunctionAnnotation> ::= <FunctionConstraint>

<ProcedureConstraint> ::= <Precondition> <Postcondition> |
<Precondition> |
<Postcondition> |
null

<FunctionConstraint> ::= <Precondition> <ReturnExpression> |
<Precondition> |
<ReturnExpression> |
null

<ProcedureSpecification> ::= rwprocedure
Identifier <FormalPart> |
rwprocedure Identifier

<FunctionSpecification> ::= rwfunction Identifier
<FormalPart> rwreturn
<TypeMark> |
rwfunction Identifier
rwreturn <TypeMark>

<FormalPart> ::= leftparen <FormalPartRep>
rightparen

<FormalPartRep> ::= <FormalPartRep> semicolon
<ParameterSpecification> |
<ParameterSpecification>

<ParameterSpecification> ::= Identifier colon <Mode>
<TypeMark>

```
<Mode> ::= rwin |
           rwin rwout |
           rwout |
           null

<PackageBody> ::= rwpackage rwbody
                  Identifier rwis <LaterDeclarativeItemRep>
                  rwend Identifier semicolon


<LaterDeclarativeItemRep> ::= <LaterDeclarativeItemRep>
                              <SubprogramBody> |
                              <SubprogramBody>

<SubprogramBody> ::= <ProcedureSpecification>
                     rwis <SubprogramImplementation> |
                     <FunctionSpecification> rwis
                     <SubprogramImplementation>

<SubprogramImplementation> ::= <InitialDeclarativeItemRep>
                               rwbegin <SequenceOfStatements>
                               rwend Identifier
                               semicolon |
                               rwbegin <SequenceOfStatements>
                               rwend Identifier
                               semicolon

<InitialDeclarativeItemRep> ::= <InitialDeclarativeItemRep>
                                <VariableDeclaration> |
                                <VariableDeclaration>

<VariableDeclaration> ::= Identifier colon <TypeMark>
                          semicolon

<SequenceOfStatements> ::= <SequenceOfStatements> <Statement> |
                           <Statement>

<Statement> ::= <SimpleStatement> |
                <CompoundStatement> |
                <ProofStatement>

<SimpleStatement> ::= <AssignmentStatement> |
                      <ProcedureCallStatement> |
                      <ExitStatement> |
                      <ReturnStatement>

<CompoundStatement> ::= <IfStatement> |
                        <LoopStatement>

<ProofStatement> ::= <AssertStatement>

<AssignmentStatement> ::= <Name> becomes <Expression>
                          semicolon

<ProcedureCallStatement> ::= <Name> semicolon

<ExitStatement> ::= rwexit semicolon

<ReturnStatement> ::= rwreturn <Expression>
                      semicolon
```

<IfStatement> ::= rwif <Condition> rwthen
<SequenceOfStatements> <ElsePart>
rwend rwif semicolon

<ElsePart> ::= rwelse <SequenceOfStatements> |
null

<LoopStatement> ::= <LoopStatementOpt> rwloop
<SequenceOfStatements> rwend
rwloop semicolon

<LoopStatementOpt> ::= <IterationScheme> |
null

<IterationScheme> ::= rwwhile <Condition> |
rwfor <LoopParameterSpecification>

<LoopParameterSpecification> ::= Identifier rwin
<TypeMark>
rwrange <Arange> |
Identifier rwin <TypeMark>

<Condition> ::= <Expression>

<Expression> ::= <Relation> |
<Relation> rwand <ExpressionRep1> |
<Relation> rwor <ExpressionRep3>

<ExpressionRep1> ::= <ExpressionRep1> rwand <Relation> |
<Relation>

<ExpressionRep3> ::= <ExpressionRep3> rwor <Relation> |
<Relation>

<Relation> ::= <SimpleExpression> |
<SimpleExpression> <RelationalOperator>
<SimpleExpression>

<RelationalOperator> ::= equals |
notequal |
lessthan |
lessorequal |
greaterthan |
greaterorequal

<SimpleExpression> ::= <SimpleExpression> <BinaryAddingOperator>
<Term> |
<SimpleExpressionOpt>

<BinaryAddingOperator> ::= plus |
minus

<SimpleExpressionOpt> ::= <UnaryAddingOperator> <Term> |
<Term>

<UnaryAddingOperator> ::= plus |
minus

<Term> ::= *<Term>* *<MultiplyingOperator>* *<Factor>* |
<Factor>

<MultiplyingOperator> ::= *multiply* |
divide

<Factor> ::= *<Primary>* |
<Primary> *doublestar* *<Primary>* |
rwnot *<Primary>*

<Primary> ::= *Integernumber* |
<Name> |
leftparen *<Expression>*
rightparen |
<Name> *Attributeident*

<TypeMark> ::= *Identifier*

<DottedSimpleName> ::= *Identifier*

<SimpleName> ::= *Identifier*

<Name> ::= *<SimpleName>* |
<Name> *leftparen* *<PositionalArgumentAssociation>*
rightparen

<PositionalArgumentAssociation> ::= *<PositionalArgumentAssociation>*
comma *<Expression>* |
<Expression>

<DependencyRelation> ::= *annotationstart* *rwderives*
annotationend

<AssertStatement> ::= *proofcontext* *rwassert*
annotationend

<Precondition> ::= *annotationstart* *rwpre*
annotationend

<Postcondition> ::= *annotationstart* *rwpost*
annotationend

<ReturnExpression> ::= *annotationstart* *rwreturn*
annotationend

Appendix G

Program Analysis Methods

G.1 Introduction

As described in Chapter 7, our program analysis heuristics are expressed through program analysis methods and abstract predicate satisfiers. Each of the program analysis methods are detailed in this chapter.

G.2 Method: scope

This method discovers those variables that are in scope at each edge of the control flow-graph.

G.2.1 Property Type

The property type for this method is shown in Figure G.1. Those variables in scope are associated with the value *inscope*.

Address $\mapsto$ Property
[*scope*, <*Var*>] $\mapsto$ *inscope*
Definitions
<*Var*> ::= <*subprogram variable*>

Figure G.1: Property type for scope

G.2.2 Route

The structured block corresponding to the entire subprogram is retrieved, and followed in sequence. Thus, with the exception of loop merge nodes, a node is visited after all of its leading nodes have been visited.

G.2.3 Property Operations

Entry

$$\boxed{entry}$$
$$[scope, <Var>] \mapsto <Scope>_{out}$$

Each variable has either static or dynamic scope. Variables with static scope are always in scope. These correspond to subprogram parameters or local declarations. The simplified package information is queried to identify each variable with static scope as *StaticVar*, setting their output properties as:

$$[scope, StaticVar] \mapsto inscope \tag{G.1}$$

Variables with dynamic scope are sometimes in scope. At subprogram entry, dynamically scoped variables are never in scope.

Assignment

$$[scope, <Var>] \mapsto <Scope>_{in}$$
$$\boxed{assign(\ldots)}$$
$$[scope, <Var>] \mapsto <Scope>_{out}$$

Assignment does not affect the scope of variables. The input properties are copied as the output properties.

EnterScope

$$[scope, <Var>] \mapsto <Scope>_{in}$$
$$\boxed{enterScope(VarRef)}$$
$$[scope, <Var>] \mapsto <Scope>_{out}$$

Entering scope brings variable *VarRef* into scope. The input properties are copied as the output properties. Further, a property is introduced for the additional variable in scope as:

$$[scope, VarRef] \mapsto inscope \tag{G.2}$$

ExitScope

$$[scope, <Var>] \mapsto <Scope>_{in}$$
$$\boxed{exitScope(VarRef)}$$
$$[scope, <Var>] \mapsto <Scope>_{out}$$

Exiting scope puts variable *VarRef* out of scope. The input properties are copied as the output properties, excluding the property associated with *VarRef*.

Branch or Loop Branch

$$[scope, <Var>] \mapsto <Scope>_{in}$$
$$\boxed{branch(\dots) \vee loopBranch(\dots)}$$
$$[scope, <Var>] \mapsto <Scope>_{out}^{true} \qquad [scope, <Var>] \mapsto <Scope>_{out}^{false}$$

A branch does not affect the scope of variables. The input properties are copied as the output properties for both the true and false edges.

Merge

$$[scope, <Var>] \mapsto <Scope>_{in}^{1} \dots [scope, <Var>] \mapsto <Scope>_{in}^{n}$$
$$\boxed{merge}$$
$$[scope, <Var>] \mapsto <Scope>_{out}$$

A merge does not affect the scope of variables. Variables entering scope must exit scope on the same path, thus the input properties on each edge must be the same. The consistent input properties are copied as the output properties.

Loop Merge

$$[scope, <Var>] \mapsto <Scope>_{in}^{entry} \qquad [scope, <Var>] \mapsto <Scope>_{in}^{return}$$
$$\boxed{loopMerge(\dots)}$$
$$[scope, <Var>] \mapsto <Scope>_{out}$$

A loop merge does not affect the scope of variables. When encountering the loop merge, there will be no properties associated with the return edge. The input properties on the entry edge are copied as the output properties.

G.3 Method: update

This method discovers two related properties of variables. Firstly, the method identifies subprogram edges where variables have been fully assigned. Secondly, the method identifies the assignment nodes that may have contributed to the currently assigned value.

G.3.1 Property Type

The property type for this method is shown in Figure G.2. Each variable is associated with its assigned and modified status. The status *assigned* indicates that the variable has been fully assigned, while *unassigned* indicates that the variable has not been fully assigned. The status *ambiguous* indicates that the assignment status can not be stated categorically. The modified status lists every node which may have contributed to the assigned status.

Address ↦ Property
[*update*, <*Var*>] ↦ <*Update*>
Definitions
<*Var*> ::= <*subprogram variable*>
<*Update*> ::= (<*Assigned*>, <*Modified*>) <*Assigned*> ::= *assigned* | *unassigned* | *ambiguous* <*Modified*> ::= <*NodeIdList*> <*NodeIdList*> ::= [] | [*NodeId* | <*NodeIdList*>]

Figure G.2: Property type for update

G.3.2 Route

The structured block corresponding to the entire subprogram is retrieved, and followed in sequence. Where a loop block is encountered, the path around the loop is followed twice[1]. Consequently, at least once, every node will be visited after all of its leading nodes have been visited.

G.3.3 Property Operations

Entry

$$\boxed{\mathit{EntryNodeId} : \mathit{entry}}$$

$$[\mathit{update}, <\mathit{Var}>] \mapsto <\mathit{Update}>_{\mathit{out}}$$

[1] In practice, this is achieved by naively duplicating the path around each loop block, regardless of its nesting. Thus, loops nested at depth d are actually itterated 2^d times. While this is clearly inefficient, it does not affect the result of the method.

The scope method and the simplified package information are queried to identify the assigned status of every variable in scope. All input parameter variables *InVarRef* must have an assigned value. Their output properties are set as follows:

$$[update, InVarRef] \mapsto (assigned, [EntryNodeId]) \tag{G.3}$$

All other variables *NotInVarRef* are unassigned, with no modification history. Their output properties are set as:

$$[update, NotInVarRef] \mapsto (unassigned, []) \tag{G.4}$$

Assignment

$$[update, <Var>] \mapsto <Update>_{in}$$
$$\boxed{AssigmentNodeId : assign(LValueExp, RValueExp)}$$
$$[update, <Var>] \mapsto <Update>_{out}$$

The variable modified by *LValueExp* is extracted as *VarRef*. The input properties are copied as the output properties, excluding the property associated with *VarRef*. The input property for *VarRef* will take the following general form:

$$[update, VarRef] \mapsto (<Assigned>_{in}, <Modified>_{in}) \tag{G.5}$$

Where *LValueExp* is a whole variable, then *VarRef* is fully assigned at this node. Its output property is set as:

$$[update, VarRef] \mapsto (assigned, [AssigmentNodeId]) \tag{G.6}$$

Where *LValueExp* is an index of an array, then only a portion of *VarRef* is assigned at this node. Its output property is set as follows:

$$[update, VarRef] \mapsto (<Assigned>_{in}, [AssigmentNode \mid <Modified>_{in}]) \tag{G.7}$$

Note that the complete assignment of an array, through cumulative updates, is not detected.

EnterScope

$$[update, <Var>] \mapsto <Update>_{in}$$
$$\boxed{enterScope(VarRef)}$$
$$[update, <Var>] \mapsto <Update>_{out}$$

Entering scope brings an additional variable *VarRef* into scope. The input properties are copied as the output properties and a property is introduced for the additional variable. As *VarRef* has just entered scope it is unassigned, with no modification history. Its output property is set as follows:

$$[update, VarRef] \mapsto (unassigned, []) \tag{G.8}$$

ExitScope

$$[update, <Var>] \mapsto <Update>_{in}$$
$$\boxed{exitScope(VarRef)}$$
$$[update, <Var>] \mapsto <Update>_{out}$$

Exiting scope removes the variable *VarRef* from scope. The input properties are copied as the output properties, excluding the property associated with *VarRef*.

Branch or Loop Branch

$$[update, <Var>] \mapsto <Update>_{in}$$
$$\boxed{branch(\ldots) \vee loopBranch(\ldots)}$$
$$[update, <Var>] \mapsto <Update>_{out}^{true} \qquad [update, <Var>] \mapsto <Update>_{out}^{false}$$

A branch does not update any variables. The input properties are copied as the output properties for both the true and false edges.

Merge

$$[update, <Var>] \mapsto <Update>_{in}^{1} \ldots [update, <Var>] \mapsto <Update>_{in}^{n}$$
$$\boxed{merge}$$
$$[update, <Var>] \mapsto <Update>_{out}$$

The input properties associated with each variable are merged to generate the output property for the variable. If every input property has the same assigned status, then this consistent status is retained. Otherwise, the assigned status is set as *ambiguous*. The modification lists are appended, deleting duplicates, indicating that any of the input branches may have been traversed.

Loop Merge

$$[update, <Var>] \mapsto <Update>_{in}^{entry} \qquad [update, <Var>] \mapsto <Update>_{in}^{return}$$

$$\boxed{loopMerge(\ldots)}$$

$$[update, <Var>] \mapsto <Update>_{out}$$

The input properties associated with each variable are merged to generate the output property for the variable. As loops are itterated twice, the loop merge will be visited on more than one occasion. Where encountered the first time, no properties will be available on the return edge. In this case, the input properties on the entry edge are copied as the output properties. Where the loop merge is encountered again, properties will now be available on the return edge. In this case input properties are merged in exactly the same manner as the merge node above. Note that the properties always stabilise following the second iteration.

G.3.4 Example

An example is given to illustrate the behaviour of this method. Consider the CheckSum subprogram shown in Figure G.3. The subprogram sums the first to eighth elements of an array and stores the result in the zeroth element of the array. For program analysis, the subprogram is translated into a control flowgraph as shown in Figure G.4.

```
package CheckSum_Package is
  subtype AE_T is Integer range 0..100;
  subtype AR1_T is Integer range 0..8;
  subtype AR2_T is Integer range 1..AR1_T'Last;
  type A_T is array (AR1_T) of AE_T;
  procedure CheckSum(A: in out A_T);
  --# derives A from A;
end CheckSum_Package;
```

```
package body CheckSum_Package is
  procedure CheckSum(A: in out A_T)
  is
    C: Integer;
  begin
    C:=0;
    for I in AR2_T loop
      --# assert true;
      C:=C+A(I);
    end loop;
    A(0):=C;
  end CheckSum;
end CheckSum_Package;
```

Figure G.3: CheckSum subprogram

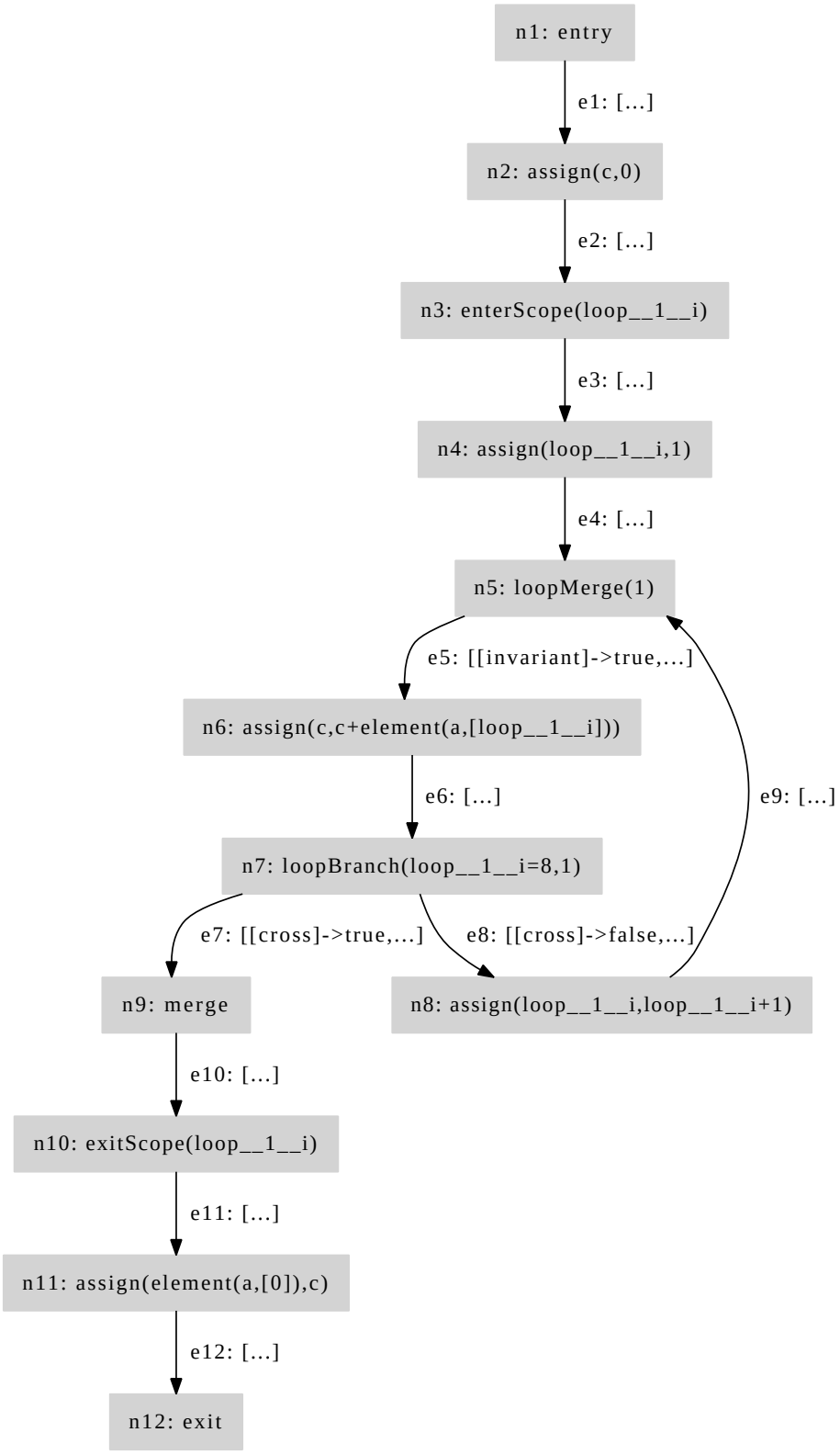


Figure G.4: CheckSum control flowgraph

The route visits every subprogram node in sequence, iterating each loop twice. The route is retrieved as:

$$[n1, n2, n3, n4, n5, n6, n7, n8, \\ n5, n6, n7, n8, n9, n10, n11, n12] \tag{G.9}$$

To begin, update properties are distributed to the start of the loop, as shown in Figure G.5. The route starts at the subprogram entry node (*n1*). The array variable *a* is a parameter of mode *inout*, thus it is initially assigned. Variable *c* is a local variable that is initially unassigned. Next, assignment node (*n2*) assigns to *c*. The next node (*n3*) brings variable *i* into scope, which is initially unassigned. Next, assignment node (*n4*) assigns to *i*.

Node *n1*	*entry*
Edge *e1*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (unassigned, [])$
Node *n2*	$assign(c, 0)$
Edge *e2*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n2])$
Node *n3*	$enterScope(i)$
Edge *e3*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n2]),$ $[update, i] \mapsto (unassigned, [])$
Node *n4*	$assign(i, 0)$
Edge *e4*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n2]),$ $[update, i] \mapsto (assigned, [n4])$

The table format above is used throughout this chapter to describe the transformations seen to properties on traversing a path through a control flowgraph. The table lists every node and edge encountered on the path. The first column identifies the node or edge being described. Where describing a node, the operation of the node is shown. Where describing an edge, the relevant properties held at the edge are shown. Every node is preceded by its input edges and followed by its output edges. To highlight these transitions, the node descriptions are shaded.

Figure G.5: update on CheckSum: Reaching loop

At this stage, update properties are distributed around the loop for the first time, as shown in Figure G.6. As the loop merge node (*n5*) is reached for the first time, no update properties exist on the edge returning from the loop. Thus, following the loop merge node, the update properties for all variables are unchanged. Next, the assignment node (*n6*) assigns to *c*. Next, a loop branch node (*n7*) is encountered. As branches do not affect update properties, the properties are unchanged on the both the true and false edges. The loop iteration is completed with the assignment node (*n8*) which assigns to *i*.

At this stage, update properties are distributed around the loop for the second time, as shown in Figure G.7. At the loop merge node, update properties now exist on the edge returning from the loop. Update properties on the edge arriving at and returning from the loop are merged. Update properties for variables *c* and *i* now list every potential assignment point. These properties are then distributed in the same manner as the first

Edge *e4*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n2]),$ $[update, i] \mapsto (assigned, [n4])$
Edge *e9*	$\emptyset$
Node *n5*	$loopMerge(1)$
Edge *e5*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n2]),$ $[update, i] \mapsto (assigned, [n4])$
Node *n6*	$assign(c, c + element(a, [i]))$
Edge *e6*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4])$
Node *n7*	$loopBranch(i = 8, 1)$
Edge *e7*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4])$
Edge *e8*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4])$
Node *n8*	$assign(i, i + 1)$
Edge *e9*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n8])$

Figure G.6: update on CheckSum: First loop iteration

loop iteration. Note that the update properties on the edge returning from the loop match those from the first iteration. Such stabilisation always occurs, thus additional iterations are not required.

Edge *e4*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n2]),$ $[update, i] \mapsto (assigned, [n4])$
Edge *e9*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n8])$
Node *n5*	$loopMerge(1)$
Edge *e5*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n2, n6]),$ $[update, i] \mapsto (assigned, [n4, n8])$
Node *n6*	$assign(c, c + element(a, [i]))$
Edge *e6*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4, n8])$
Node *n7*	$loopBranch(i = 8, 1)$
Edge *e7*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4, n8])$
Edge *e8*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4, n8])$
Node *n8*	$assign(i, i + 1)$
Edge *e9*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n8])$

Figure G.7: update on CheckSum: Second loop iteration

Next, update properties are distributed from the edge leaving the loop to the end of the subprogram, as shown in Figure G.8. The merge node (*n9*) contains a single input edge, thus update properties following the merge are unchanged. The following node (*n10*) puts variable *i* out of scope. Finally, the assignment node (*n11*) assigns to the zeroth element of array *a*. This partial assignment leads to the list of potential modifications being extended. The route is now complete, marking the completion of the method.

Edge *e7*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4, n8])$
Node *n9*	*merge*
Edge *e10*	$[update, a] \mapsto (assigned, [n1]), [update, c] \mapsto (assigned, [n6]),$ $[update, i] \mapsto (assigned, [n4, n8])$
Node *n8*	*exitScope*(*i*)
Edge *e11*	$[update, a] \mapsto (assigned, [n1])[update, c] \mapsto (assigned, [n6])$
Node *n11*	$assign(element(a, [0]), c)$
Edge *e12*	$[update, a] \mapsto (assigned, [n1, n11]), [update, c] \mapsto (assigned, [n6])$

Figure G.8: update on CheckSum: Leaving loop

G.4 Method: context

This method discovers different structural contexts that exist within the subprogram.

G.4.1 Property Type

The property type for this method is shown in Figure G.9. A single property is held at each edge, describing its context through a list of tags. The context at edge *e1* also holds at edge *e2* if all of the tags at *e1* are also at *e2*.

Address ↦ Property
[*context*] ↦ <*TagList*>
Definitions
<*TagList*> ::= [] | [*Tag* | <*TagList*>]

Figure G.9: Property type for method context

G.4.2 Route

The structured block corresponding to the entire subprogram is retrieved, and followed in sequence. Thus, with the exception of loop merge nodes, a node is visited after all of its leading nodes have been visited.

G.4.3 Property Operations

Entry

$$\boxed{entry}$$

$$[context] \mapsto <TagList>_{out}$$

A unique tag is created to describe the context of the subprogram as *SubprogramTag*, setting the output property as:

$$[context] \mapsto [SubprogramTag] \tag{G.10}$$

Assignment, EnterScope and ExitScope

$$[context] \mapsto <TagList>_{in}$$

$$\boxed{assignment(\ldots) \vee enterScope(\ldots) \vee exitScope(\ldots)}$$

$$[context] \mapsto <TagList>_{out}$$

Assignment and entering and exiting scope do not affect context. Thus, the input property is copied as the output property.

Branch or Loop Branch

$$[context] \mapsto <TagList>_{in}$$
$$\boxed{branch(\ldots) \vee loopBranch(\ldots)}$$
$$[context] \mapsto <TagList>_{out}^{true} \qquad [context] \mapsto <TagList>_{out}^{false}$$

Following a branch, the context is extended. A unique tag is created to describe the context of the true and false edges as *TrueTag* and *FalseTag* respectively. The output property for the true edge is set as:

$$[context] \mapsto [TrueTag \mid <TagList>_{in}] \tag{G.11}$$

While the output property for the false edge is set as:

$$[context] \mapsto [FalseTag \mid <TagList>_{in}] \tag{G.12}$$

Merge

$$[context] \mapsto <TagList>_{in}^{1} \ldots [context] \mapsto <TagList>_{in}^{n}$$
$$\boxed{merge}$$
$$[context] \mapsto <TagList>_{out}$$

Following a merge, the context is contracted. The output property is the intersection of the tags at every input property:

$$[context] \mapsto <TagList>_{in}^{1} \cap \ldots <TagList>_{in}^{n} \tag{G.13}$$

Loop Merge

$$[context] \mapsto <TagList>_{in}^{entry} \qquad [context] \mapsto <TagList>_{in}^{return}$$
$$\boxed{loopMerge(\ldots)}$$
$$[context] \mapsto <TagList>_{out}$$

Only one loop iteration is considered. Thus, on encountering the loop merge node, there will be no context property associated with the return branch. The single input property is copied as its output property.

G.4.4 Example

An example is given to illustrate the behaviour of this method. Consider the FindIndex subprogram shown in Figure G.10. The subprogram searches through an array to find the first occurrence of a requested element. Where the element is found its index is returned, otherwise zero is returned. For program analysis, the subprogram is translated into a control flowgraph as shown in Figure G.11.

```
package FindIndex_Package is
  subtype AR_T is Integer range 1..10;
  subtype EAR_T is Integer range 0..10;
  type A_T is array (AR_T) of Integer;
  procedure FindIndex(A: in A_T; S: in Integer;
                      R: out EAR_T);
  --# derives R from A,S;
end FindIndex_Package;
```

```
package body FindIndex_Package is
  procedure FindIndex(A: in A_T; S: in Integer;
                      R: out EAR_T)
  is
  begin
    R:=0;
    for I in AR_T loop
      --# assert true;
      if A(I)=S then
        R:=I;
        exit;
      end if;
    end loop;
  end FindIndex;
end FindIndex_Package;
```

Figure G.10: FindIndex subprogram

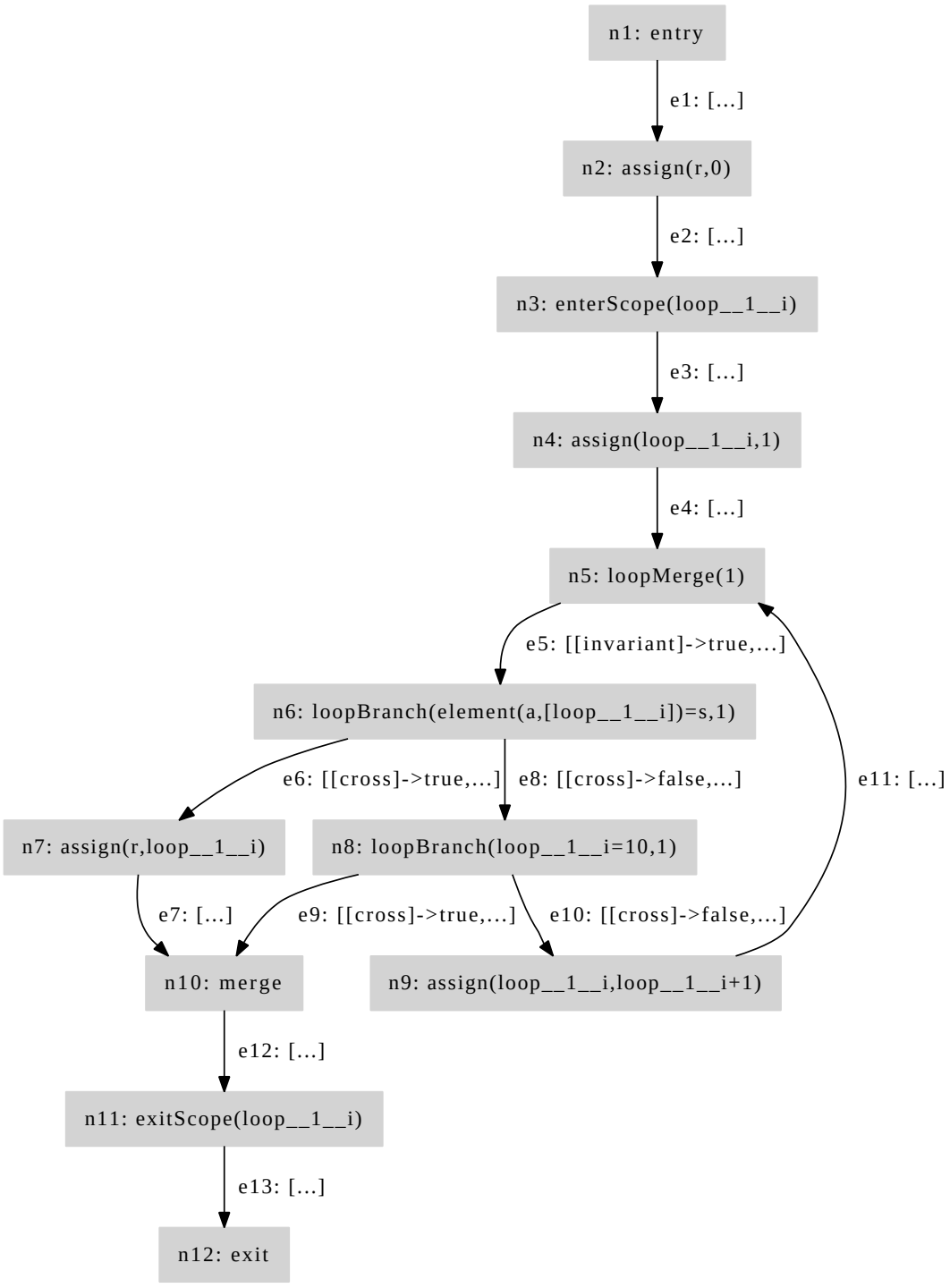


Figure G.11: FindIndex control flowgraph

The route visits every subprogram node in sequence. The route is retrieved as:

$$[n1, n2, n3, n4, n5, n6, n8, n9, n7, n10, n11, n12] \tag{G.14}$$

To begin, context properties are distributed to the start of the loop, as shown in Figure G.12. The route starts at the subprogram entry node (*n1*). The initial context is indicated via the tag *e*. The next three nodes encountered are an assignment node (*n2*), an enter scope node (*n3*) and another assignment node (*n4*). These nodes do not modify the initial context property.

Node *n1*	*entry*
Edge *e1*	$[context] \mapsto [e]$
Node *n2*	$assign(r, 0)$
Edge *e2*	$[context] \mapsto [e]$
Node *n3*	$enterScope(i)$
Edge *e3*	$[context] \mapsto [e]$
Node *n4*	$assign(i, 1)$
Edge *e4*	$[context] \mapsto [e]$

Figure G.12: context on FindIndex: Reaching loop

At this stage, context properties are distributed around the loop, as shown in Figure G.13. The loop merge node (*n5*) does not modify context properties. At the first loop branch node (*n6*) the context properties associated with the true and false edges are extended with the tags *b1t* and *b1f* respectively. Similarly, at the second loop branch node (*n8*) the context properties associated with the true and false edges are extended with the tags *b2t* and *b2f* respectively. The loop iteration is completed at assignment node (*n9*), which does not modify context properties.

Edge *e4*	$[context] \mapsto [e]$
Edge *e11*	$\emptyset$
Node *n5*	$loopMerge(1)$
Edge *e5*	$[context] \mapsto [e]$
Node *n6*	$loopBranch(element(a, [i]) = s, 1)$
Edge *e6*	$[context] \mapsto [e, b1t]$
Edge *e8*	$[context] \mapsto [e, b1f]$
Node *n8*	$loopBranch(i = 10, 1)$
Edge *e9*	$[context] \mapsto [e, b1f, b2t]$
Edge *e10*	$[context] \mapsto [e, b1f, b2f]$
Node *n9*	$assign(i, i + 1)$
Edge *e11*	$[context] \mapsto [e, b1f, b2f]$

Figure G.13: context on FindIndex: Loop iteration

Next, context properties are distributed from the edges leaving the loop to the end of the subprogram, as shown in Figure G.14. Leaving the loop via the first branch node (*n6*), the assignment node (*n7*) is encountered, which does not modify context properties. There are no nodes to consider on the path leaving the second branch node (*n8*). The

merge node (*n10*) contracts context properties. Only the subprogram tag *e* is common to the merged context properties. Finally, the exit scope node (*n11*) is reached, making no modifications to context properties. The route is now complete, marking the completion of the method.

Edge *e6*	$[context] \mapsto [e, b1t]$
Node *n7*	$assign(r, i)$
Edge *e7*	$[context] \mapsto [e, b1t]$
Edge *e9*	$[context] \mapsto [e, b1f, b2t]$
Node *n10*	*merge*
Edge *e12*	$[context] \mapsto [e]$
Node *n11*	$exitScope(i)$
Edge *e13*	$[context] \mapsto [e]$

Figure G.14: context on FindIndex: Leaving loop

G.5 Method: type

This method introduces properties stating that assigned variables are within their type.

G.5.1 Property Type

The property type for this method is shown in Figure G.15. Each variable is associated with its corresponding type constraint.

Address $\mapsto$ Property
[*type*, <*Var*>] $\mapsto$ <*TypeConstraint*>
Definitions
<*Var*> ::= <*subprogram variable*>

Figure G.15: Property type for method type

G.5.2 Route

The structured block corresponding to the entire subprogram is retrieved. Every subprogram node is visited, in any order.

G.5.3 Property Operations

Every Node

$$[type, <Var>] \mapsto <TypeConstraint>^{1}_{in} \ldots [type, <Var>] \mapsto <TypeConstraint>^{n}_{in}$$

$$\boxed{\cdots}$$

$$[type, <Var>] \mapsto <TypeConstraint>^{1}_{out} \ldots [type, <Var>] \mapsto <TypeConstraint>^{m}_{out}$$

The same property operation is applied for every node. The input edges are always ignored. The update method and the simplified package information are queried to associate every assigned variable on the output edges with its corresponding type constraint.

G.6 Method: transient

This method discovers transient properties that are preserved for sections of the subprogram. The preservation of the transient properties are calculated from the assignment status of variables and the structural contexts that exist in the subprogram.

G.6.1 Property Type

The property type for this method is shown in Figure G.16. Multiple properties may be held at each edge. Each property contains a constraint alongside two conditions which, if preserved, mean that the constraint continues to hold. The first condition specifies required update properties for selected variables. The second condition specifies the required structural context.

Address ↦ Property
[*transient*] ↦ <*Transient*>
Definitions
<*Transient*> ::= (*Constraint*, <*UpdateList*>, <*TagList*>) <*UpdateList*> ::= [] | [(<*Var*>, <*Update*>) | <*UpdateList*>] <*Var*> ::= <*subprogram variable*> <*TagList*> ::= defined in property type for context method <*Update*> ::= defined in property type for update method

Figure G.16: Property type for method transient

G.6.2 Route

The structured block corresponding to the entire subprogram is retrieved, and followed in sequence. Thus, with the exception of loop merge nodes, a node is visited after all of its leading nodes have been visited.

G.6.3 Property Operations

Every Node

$$[transient] \mapsto <Transient>^{1}_{in} \ldots [transient] \mapsto <Transient>^{n}_{in}$$

$$\boxed{\cdots}$$

$$[transient] \mapsto <Transient>^{1}_{out} \ldots [transient] \mapsto <Transient>^{m}_{out}$$

This property operation is applied at every node to distribute previously introduced transient properties. Each transient property on an input edge is investigated individually. The property is copied to an output edge if its two conditions are satisfied. Firstly, the

recorded update properties for selected variables must match those on the output edge. Secondly, the recorded context must be available on the output edge.

Assignment

$$[transient] \mapsto <Transient>_{in}$$
$$\boxed{assign(LValueExp, RValueExp)}$$
$$[transient] \mapsto <Transient>_{out}$$

Following assignment, a transient property may be introduced on the output edge. Where the variable modified through *LValueExp* is not referenced in the assigned expression *RValueExp* then the assignment can be trivially expressed as an equality. This observation is exploited to introduce the following transient property:

$$[transient] \mapsto (LValueExp = RValueExp, <UpdateList>, <TagList>) \qquad \text{(G.15)}$$

The update method is queried, pairing each variable in *LValueExp* and *RValueExp* with its update property as <*UpdateList*>. Further, the context method is queried to determine the context property following the assignment as <*TagList*>.

Branch or Loop Branch

$$[transient] \mapsto <Transient>_{in}$$
$$\boxed{branch(ConditionExp) \vee loopBranch(ConditionExp, \ldots)}$$
$$[transient] \mapsto <Transient>_{out}^{true} \qquad [transient] \mapsto <Transient>_{out}^{false}$$

Following a branch, a transient property is introduced on the true edge as:

$$[transient] \mapsto (ConditionExp, <UpdateList>, <TagList>) \qquad \text{(G.16)}$$

While a transient property is introduced on the false edge as:

$$[transient] \mapsto (\neg ConditionExp, <UpdateList>, <TagList>) \qquad \text{(G.17)}$$

The update method is queried to pair each variable referenced in *ConditionExp* with its corresponding update property as <*UpdateList*>. Further, the context method is queried to determine the context property on the output edge as <*TagList*>.

G.6.4 Example

An example is given to illustrate the behaviour of this method. Consider the IndexInitArray subprogram shown in Figure G.17. The subprogram initialises an array such that the element at each index equals this index. For program analysis, the subprogram is translated into a control flowgraph as shown in Figure G.18.

```
package IndexInitArray_Package is
  subtype AR_T is Integer range 0..1000;
  type A_T is array (AR_T) of AR_T;
  procedure IndexInitArray(A: in out A_T);
  --# derives A from A;
end IndexInitArray_Package;
```

```
package body IndexInitArray_Package is
  procedure IndexInitArray(A: in out A_T)
  is
  begin
    for I in AR_T loop
      A(I):=I;
    end loop;
  end IndexInitArray;
end IndexInitArray_Package;
```

Figure G.17: IndexInitArray subprogram

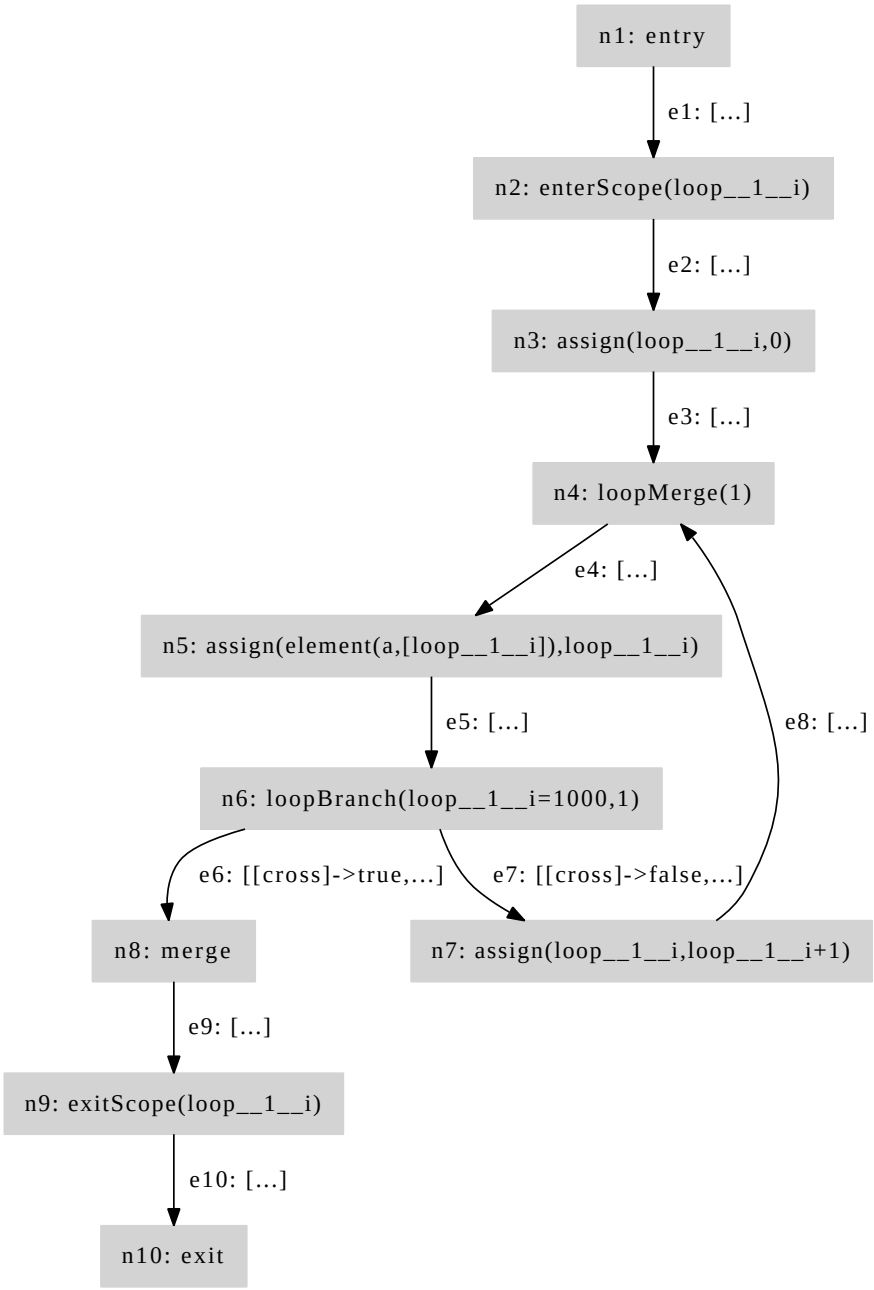


Figure G.18: IndexInitArray control flowgraph

The route visits every subprogram node in sequence. The route is retrieved as:

$$[n1, n2, n3, n4, n5, n6, n7, n8, n9, n10] \tag{G.18}$$

The distribution of transient properties on reaching and iterating around the loop are shown in Figure G.19. The subprogram entry node (*n1*) and the enter scope node (*n2*) do not affect transient properties. The assignment node (*n3*), leads to the introduction of a transient property. Following the loop merge node (*n4*) the update property for variable *i* changes, reflecting the assignment seen on loop iterations. Consequently, the conditions associated with the transient property introduced at (*n3*) no longer hold and the transient property is removed. The following assignment node (*n5*) leads to the introduction of another transient property. Next, a branch node (*n6*) is encountered, introducing transient properties for each departing edge. The subsequent assignment node (*n7*) does not lead to the introduction of a transient property, as the modified variable *i* is also referenced in the assigned expression. Following this assignment node the update property for variable *i* changes. Consequently, the conditions associated with the transient properties introduced at (*n5*) and the false edge of (*n6*) no longer hold and these transient properties are removed.

Node *n1*	*entry*
Edge *e1*	$[update, a] \mapsto (assigned, [n1]), [context] \mapsto [e]$
Node *n2*	$enterScope(i)$
Edge *e2*	$[update, a] \mapsto (assigned, [n1]), [update, i] \mapsto (unassigned, []), [context] \mapsto [e]$
Node *n3*	$assign(i, 0)$
Edge *e3*	$[update, a] \mapsto (assigned, [n1]), [update, i] \mapsto (assigned, [n3]), [context] \mapsto [e]$ $[transient] \mapsto (i = 0, [(i, (assigned, [n3]))], [e])$
Node *n4*	$loopMerge(1)$
Edge *e4*	$[update, a] \mapsto (assigned, [n1, n5]), [update, i] \mapsto (assigned, [n3, n7]), [context] \mapsto [e]$
Node *n5*	$assign(element(a, [i]), i)$
Edge *e5*	$[update, a] \mapsto (assigned, [n1, n5]), [update, i] \mapsto (assigned, [n3, n7]), [context] \mapsto [e]$ $[transient] \mapsto (element(a, [i]) = i, [(a, (assigned, [n1, n5])), (i, (assigned, [n3, n7]))], [e])$
Node *n6*	$loopBranch(i = 1000, 1)$
Edge *e6*	$[update, a] \mapsto (assigned, [n1, n5]), [update, i] \mapsto (assigned, [n3, n7]), [context] \mapsto [e, b1t]$ $[transient] \mapsto (element(a, [i]) = i, [(a, (assigned, [n1, n5])), (i, (assigned, [n3, n7]))], [e])$ $[transient] \mapsto (i = 1000, [(i, (assigned, [n3, n7]))], [e, b1t])$
Edge *e7*	$[update, a] \mapsto (assigned, [n1, n5]), [update, i] \mapsto (assigned, [n3, n7]), [context] \mapsto [e, b1f]$ $[transient] \mapsto (element(a, [i]) = i, [(a, (assigned, [n1, n5])), (i, (assigned, [n3, n7]))], [e])$ $[transient] \mapsto (\neg(i = 1000), [(i, (assigned, [n3, n7]))], [e, b1f])$
Node *n7*	$assign(i, i + 1)$
Edge *e8*	$[update, a] \mapsto (assigned, [n1, n5]), [update, i] \mapsto (assigned, [n7]), [context] \mapsto [e, b1f]$

Figure G.19: transient on IndexInitArray: Reaching loop

The distribution of transient properties on leaving the loop to the end of the subprogram are shown in Figure G.20. As the loop has a single exit path, the merge node (*n8*) does not restrict context, allowing transient properties to be distributed. Finally, the exit scope node (*n9*) removes *i* from scope, changing its update property and leading to all distributed properties being removed. The route is now complete, marking the completion

of the method.

Edge *e6*	$[update, a] \mapsto (assigned, [n1, n5]), [update, i] \mapsto (assigned, [n3, n7]), [context] \mapsto [e, b1t]$ $[transient] \mapsto (element(a, [i]) = i, [(a, (assigned, [n1, n5])), (i, (assigned, [n3, n7]))], [e])$ $[transient] \mapsto (i = 1000, [(i, (assigned, [n3, n7]))], [e, b1f])$
Node *n8*	*merge*
Edge *e6*	$[update, a] \mapsto (assigned, [n1, n5]), [update, i] \mapsto (assigned, [n3, n7]), [context] \mapsto [e, b1t]$ $[transient] \mapsto (element(a, [i]) = i, [(a, (assigned, [n1, n5])), (i, (assigned, [n3, n7]))], [e])$ $[transient] \mapsto (i = 1000, [(i, (assigned, [n3, n7]))], [e, b1f])$
Node *n9*	*exitScope*(*i*)
Edge *e10*	$[update, a] \mapsto (assigned, [n1, n5]), [context] \mapsto [e, b1t]$

Figure G.20: transient on IndexInitArray: Leaving loop

G.7 Method: loop_range

This method introduces properties stating that for-loop variables are within any declared range.

G.7.1 Property Type

The property type for this method is shown in Figure G.21. For-loop variables that are known to iterate between range expressions are associated with these constraints.

Address ↦ Property
[*looprange*, <*Var*>] ↦ <*RangeConstraint*>
Definitions
<*Var*> ::= <*subprogram variable*>

Figure G.21: Property type for method loop_range

G.7.2 Route

The structured block corresponding to the entire subprogram is retrieved. Every subprogram node is visited, in any order.

G.7.3 Property Operations

Every Node

$$[looprange, \langle Var \rangle] \mapsto \langle RangeConstraint \rangle^{1}_{in} \ldots$$
$$[looprange, \langle Var \rangle] \mapsto \langle RangeConstraint \rangle^{n}_{in}$$
$$\boxed{\cdots}$$
$$[looprange, \langle Var \rangle] \mapsto \langle RangeConstraint \rangle^{1}_{out} \ldots$$
$$[looprange, \langle Var \rangle] \mapsto \langle RangeConstraint \rangle^{m}_{out}$$

The same property operation is applied for every node. The input edges are always ignored. Each output edge is considered separately. The scope method and the simplified package information are queried to identify every for-loop variable in scope that has an explicit range constraint. In some situations, the range constraints can not be directly expressed as SPARK assertions. For example, the constraints may reference loop entry variables. To resolve this, a suitable goal is constructed and sent to the pa_spark_exp strategy. The type and transient properties on the output edge describe the context as hypotheses, while the range constraint forms the conclusion. Where the strategy is successful, the resulting constraint is stored on the output edge.

G.8 Method: int_constraint

This method discovers constraints for integer variables. In particular, invariant constraints are discovered for loops through the generation and solving of *recurrence relations*.

G.8.1 Property Type

Address $\mapsto$ Property
[*intconst*, <*Var*>, <*Class*>] $\mapsto$ <*Constraint*>
Definitions
<*Var*> ::= <*subprogram variable*>
<*Class*> ::= *circulate*(*SelLoopId*) | *solution* | *propagate* | *normalise*
<*Constraint*> ::= *undef* | <*Eq*> $\wedge$ <*Eq*> | <*Eq*> $\vee$ <*Eq*> | $\neg$<*Eq*> | <*boolean*>
<*Eq*> ::= <*Exp*> = <*Exp*> | <*Exp*> $\neq$ <*Exp*> | <*Exp*> < <*Exp*> | <*Exp*> $\leq$ <*Exp*> | <*Exp*> > <*Exp*> | <*Exp*> $\geq$ <*Exp*>
<*Exp*> ::= <*Exp*> $*$ $*$<*Exp*> | <*Exp*> $*$ <*Exp*> | <*Exp*> + <*Exp*> | <*Exp*> $-$ <*Exp*> | $-$<*Exp*> | <*integer*> | <*subprogram variable*~ > | <*Circulate*>(*SelLoopId*) {*where* <*Class*> = *circulate*(*SelLoopId*)} | <*Propagate*> {*where* <*Class*> $\in$ {*solution*, *propagate*}} | <*Normalise*> {*where* <*Class*> = *normalise*}
<*Circulate*>(*SelLoopId*) ::= uv_i {$i \geq 0$} | *itt*(<*subprogram variable*>, 0) | *itt*(<*Var*>, $l_{SelLoopId}$) | *itt*(<*Var*>, $l_{SelLoopId} - 1$) | l_i {*where* $i \neq$ *SelLoopId*}
<*Propagate*> ::= uv_i {$i \geq 0$} | *itt*(<*subprogram variable*>, 0) | l_i {$i \geq 1$} | <*Var*>
<*Normalise*> ::= <*subprogram variable*>

Figure G.22: Property type for method int_constraint

The property type for this method is shown in Figure G.22. Each integer variable is associated with an integer constraint. A relatively limited constraint grammar is employed to minimise complexity and remain within the capabilities of a recurrence relation solver. Nevertheless, the constraint grammar is sufficient for reasoning about many common programing constructs. The grammar references additional variables and functions, as described below:

- **Unconstrained variables** - Unconstrained variables support the elimination of expressions, as described in §G.8.2. The variables are denoted uv_i, and are introduced

for increasing values of i as required.

- **Loop iteration variables** - Loop iteration variables support the expression of invariant properties discovered through solving recurrence relations. The variables are denoted l_i, where i matches the unique number associated with a loop. These variables are implicitly zero on entry to the loop, and are implicitly increased by one at the end of each iteration.

- **Loop iteration function** - A loop iteration function is introduced to support the expression of recurrence relations. The function $itt(v, l_i)$ describes the value of variable v in loop i on the l_i^{th} iteration. For simplicity, only the first, current and previous iterations are referenced as 0, l_i and $l_i - 1$ respectively.

The grammar supports four different property classes, as described below:

- *circulate*(*SelLoopId*) - Describes the potential assignments made to an integer variable within loop *SelLoopId* through recurrence relations. Circulate properties are distributed throughout their corresponding loop. Note, however, that their corresponding loop may contain nested loops.

- *solution* - Describes invariant constraints on integer variables through solved recurrence relations. Solution properties are only associated with the edge leaving a loop merge node.

- *propagate* - Describes constraints on integer variables, including invariant constraints within loops. Propagate properties are distributed throughout the subprogram.

- *normalise* - Describes invariant properties strictly in terms of SPARK constructs. Normalise properties are only associated with edges corresponding to a loop invariant.

G.8.2 Eliminate Expressions via Unconstrained Variables

This method often requires properties to be expressed independent to other variables. For example, assume that the following two properties are known:

$$(a \geq 0) \wedge (a \leq 10) \tag{G.19}$$

$$b = 2 * a \tag{G.20}$$

Via mathematical reasoning, given (G.19), (G.20) can be expressed independently to a as:

$$(b \geq 0) \wedge (b \leq 20) \tag{G.21}$$

In general, such reasoning is difficult to automate. Instead, variables may be eliminated from properties through the introduction of unconstrained variables. Firstly, a constraint must be discovered for the variable to be eliminated. For example, if seeking to eliminate a from (G.20), then the following constraint may be found:

$$(a \geq 0) \wedge (a \leq 10) \tag{G.22}$$

While the discovery of constraints requires mathematical reasoning, the process is typically tractable. Secondly, the discovered constraint is conjoined with the original property. For example, conjoining (G.20) with (G.22) gives:

$$(b = 2 * a) \wedge ((a \geq 0) \wedge (a \leq 10)) \tag{G.23}$$

Finally, the variable to be eliminated is consistently replaced with an unconstrained variable. For example, replacing a with uv_1 in (G.23) gives:

$$(b = 2 * uv_1) \wedge ((uv_1 \geq 0) \wedge (uv_1 \leq 10)) \tag{G.24}$$

Thus, the property (G.20) is now expressed independently to a.

G.8.3 Route

Every structured block corresponding to a loop is retrieved. Sequential loops are selected from top to bottom and nested loops are selected from the innermost to the outermost. Each loop is individually analysed. The path departing from and returning to the loop merge node is iteratively followed until all variables are solved or no new solutions are discovered. Once all loops have been explored, the structured block corresponding to the entire subprogram is retrieved and followed in sequence.

G.8.4 Property Operations

Entry

$$\boxed{entry}$$
$$[intconst, <Var>, propagate] \mapsto <Constraint>_{out}$$

As the subprogram entry node is encountered after all loops have been visited, all properties must belong to the *propagate* class. The update method and the simplified package information are queried to identify the assigned status of every integer program variable in scope. Assigned variables *AssignedIntVar* are associated with an assigned value *AssignedValue*. For strictly input parameter variables the assigned value is the program variable. Otherwise the assigned value is the initial parameter variable correspond-

ing to the program variable:

$$[intconst, <Var>, propagate] \mapsto AssignedIntVar = AssignedValue \quad \text{(G.25)}$$

Unassigned variables *UnassignedIntVar* are associated with *undef*:

$$[intconst, <Var>, propagate] \mapsto UnassignedIntVar = undef \quad \text{(G.26)}$$

Assignment

$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}$$
$$\boxed{assign(LValueExp, RValueExp)}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}$$

The variable modified by *LValueExp* is extracted as *VarRef*. The input properties are copied as the output properties, excluding any property associated with *VarRef*. Where there is an input property for *VarRef* it is queried to form the initial output property as follows:

$$[intconst, <Var>, <Class>] \mapsto VarRef = RValueExp \quad \text{(G.27)}$$

The expression *RValueExp* may contain subexpressions outside the property type grammar. Some of these incompatibilities are resolved through recursively applying the following transformations:

- **Array element access** - An array element access *element*(*Array*, [*Index*]) may not be referenced in the constraint grammar. The array element access is eliminated as described in §G.8.2. The constraint for the array element access is discovered by sending a suitable goal to the pa_exp_constrain strategy. The type and transient properties on the output edge describe the context as hypotheses while the array element access forms the conclusion.

- **Program variable access** - A program variable may only be referenced in the constraint grammar in certain situations. Outside these situations, the program variable access is eliminated as described in §G.8.2. The input properties are queried to find a constraint for the program variable.

- **Bound function** - The constraint grammar does not include the *bound*(*TypeRef*) function, as introduced in §7.6.4. The bound function is eliminated as described in §G.8.2. The constraint bound function is found by adopting its declared type constraint.

Where *RValueExp* is transformed into the property type grammar, it is subsequently simplified via the pa_exp_simplify strategy. Where unsuccessful, *RValueExp* is set as *undef*.

EnterScope

$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}$$
$$\boxed{enterScope(VarRef)}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}$$

Entering a scope does not modify any variables. Thus, the input properties are copied as the output properties. As scope changes are never encountered inside loops, the properties must belong to the *propagate* class. Where *VarRef* is an integer program variable the following output property is introduced:

$$[intconst, VarRef, propagate] \mapsto VarRef = undef \tag{G.28}$$

ExitScope

$$[Class, Var] \mapsto <Constraint>_{in}$$
$$\boxed{exitScope(VarRef)}$$
$$[Class, Var] \mapsto <Constraint>_{out}$$

Exiting scope removes the variable *VarRef* from scope. The input properties are copied as the output properties, excluding any property associated with *VarRef*. As scope changes are never encountered inside loops, the properties must belong to the *propagate* class. Where *VarRef* is an integer program variable its output property is set as:

$$[intconst, VarRef, propagate] \mapsto VarRef = undef \tag{G.29}$$

Branch

$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}$$
$$\boxed{branch(\ldots)}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}^{true}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}^{false}$$

A branch does not update any variables. The input properties are copied as the output properties on both the true and false edges.

Loop Branch

$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}$$
$$\boxed{loopBranch(ConditionExp, LoopId)}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}^{true}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}^{false}$$

Different operations occur at the true and false edges, as detailed below:

- **True Edge** - Circulate properties are not distributed outside their corresponding loop. Where an input property has the following form:

$$[intconst, VarRef, circulate(LoopId)] \mapsto <Constraint>_{in} \quad (G.30)$$

  Then its output property is set as:

$$[intconst, VarRef, propagate] \mapsto VarRef = undef \quad (G.31)$$

  In all other cases, each input property is copied as the output property, following a transformation. In exiting loop *LoopId*, all references to the loop iteration variable l_{LoopId} must be removed. The loop iteration variable is eliminated as described in §G.8.2. The constraint for the loop iteration variable is discovered by sending a suitable goal to the pa_spark_exp strategy. The propagate input properties, excluding the variable under consideration, plus the type and transient properties on the output edge describe the context as hypotheses. The loop iteration variable forms the conclusion.

- **False Edge** - The false edge remains within the loop. Each input property is copied as an output property.

Merge

$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}^{1} \ldots$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}^{n}$$
$$\boxed{merge}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}$$

The input properties associated with each variable are merged to generate the output property for the variable. Where every input property associated with a variable has a defined property then these properties are disjoined. The resulting constraint is simplified

through the pa_exp_simplify strategy, and taken as the output property. Otherwise, the output property is set as *undef*.

Loop Merge (Recurrence Relation Solving)

$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}^{entry}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}^{return}$$
$$\boxed{loopMerge(LoopId)}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}$$

The input properties associated with the returning edge are inspected. Those variables of class *circulate*(*LoopId*), describe an iteration of this loop. For each such variable solved properties are sought that describe a general iteration. The stages of this process are detailed below:

- **Reject if no reference to previous iteration** - Where the circulate property is *undef* then no solution can be found. Further, where a property makes no reference to the previous iteration, no solution can be found. This situation arises where a variable is assigned a distinct value on each iteration. For example, a temporary variable might be present within a loop, but be overwritten with unrelated values on each iteration.

- **Complete the iteration** - The circulate property resides on the edge returning to the loop merge node. As initialisation took place on the edge leaving the loop merge node, the property does not yet describe a full iteration. It is necessary to carry the constraint across the loop junction to complete the iteration. No program variables are modified as the loop junction is traversed. However, the loop iteration variable l_{LoopId} is implicitly incremented. Thus, to retain the same meaning, every occurrence of l_{LoopId} in the known constraint is decremented.

- **Disjunctive normal form** - The circulate property will be expressed through a nested conjunction and disjunction of expressions. To ease analysis, the property is converted into disjunctive normal form via the pa_disj_norm_form strategy.

- **Extract extreme recurrence relations** - Each disjunct is processed individually, generating a number of *extreme recurrence relations*. These recurrence relations seek to describe the extreme edges of a constraint. Extreme recurrence relations are trivially generated by considering every combination of the lower and upper constraints of every bounded expression. The simplistic approach is only accurate where constraints are linear. Nevertheless, the technique is effective and supports the analysis of many realistic problems.

- **Solve extreme recurrence relations** - Every extreme recurrence relation is solved[2] via the recurrence relation solver PURRS [PUR]. The property type for this method is directly supported by the PURRS grammar.

- **Bounding extreme recurrence relations** - Each solved extreme recurrence relation describes a potential constraint. A lowermost and uppermost recurrence relation is sought that bounds the values of every other constraint. This is achieved by sorting the solved extreme recurrence relations through numerical analysis. A parameter set is generated, associating each parameter with a random value. Using this parameter set, each recurrence relation is evaluated and associated with its numerical solution. Based on this solution, the recurrence relations are sorted to identify the extreme bounding solutions. The process is repeated several times and must consistently produce the same result. The simplistic approach is only accurate where the constraints are linear. Nevertheless, the technique provides an effective analysis with no reasoning overhead.

Where successful, the result is associated with the *solved* class and taken as the output property.

Loop Merge (Iteration)

$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}^{entry}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{in}^{return}$$
$$\boxed{loopMerge(LoopId)}$$
$$[intconst, <Var>, <Class>] \mapsto <Constraint>_{out}$$

The update method and the simplified package information are queried to identify every assigned integer program variable. The input properties and output property are inspected in determining the output property for each variable as below:

- **Solution present** - A solution for this variable is present on the output property. The solution is copied as the output property for the variable. Where the variable has class *circulate*(*LoopId*), this loop is under investigation. In this case the variable is now solved, and the *propagate* class is adopted. Otherwise, an outer context is under investigation. In this case, the class associated with the variable is preserved. Further, any initial values referenced in the solution are replaced with the property associated with the variable on the entry edge.

[2] In practice, to minimise implementation effort, this method does not communicate directly with PURRS. Instead a look-up table is maintained, describing the capabilities of PURRS for each recurrence relation encountered.

- **No solution present** - A solution for this variable is not present on the output property. Where the variable has class *circulate*(*LoopId*), this loop is under investigation. In this case, the variable is initialised to its value on the previous iteration as:

$$[intconst, <Var>, <circulate>(LoopId)] \mapsto \\ itt(<Var>, l_{LoopId}) = <itt>(Var, l_{LoopId} - 1) \tag{G.32}$$

  Otherwise, the outer context is under investigation. The class associated with this variable is preserved. As no solution is present, the output property *undef* is adopted.

Every Node

$$[intconst, <Var>, <Class>] \mapsto <Constraint>^{1}_{in} \ldots \\ [intconst, <Var>, <Class>] \mapsto <Constraint>^{n}_{in} \\ \boxed{\cdots} \\ [intconst, <Var>, <Class>] \mapsto <Constraint>^{1}_{out} \ldots \\ [intconst, <Var>, <Class>] \mapsto <Constraint>^{m}_{out}$$

This property operation is applied for every node. The input edges are always ignored. Each output edge is considered separately. Where the output edge corresponds to an invariant, each variable associated with the *propagate* class is translated as a property of the *normalise* class. The normalised property is discovered by sending a suitable goal to the pa_spark_exp strategy. The propagate output properties, excluding the variable under consideration, plus the type and transient properties on the output edge describe the context as hypotheses. The *propagate* property associated with the variable forms the conclusion.

G.8.5 Example

An example is given to illustrate the behaviour of this method. Consider the FilterInteger subprogram shown in Figure G.23. The subprogram sums all of the elements in an array that lie between 0 and 100. For program analysis, the subprogram is translated into a control flowgraph as shown in Figure G.24.

```
package FilterInteger_Package is
  subtype AR_T is Integer range 0..9;
  type A_T is array (AR_T) of Integer;
  procedure FilterInteger(A: in A_T; R: out Integer);
  --# derives R from A;
end FilterInteger_Package;
```

```
package body FilterInteger_Package is
  procedure FilterInteger(A: in A_T; R: out Integer)
  is
  begin
    R:=0;
    for I in AR_T loop
      --# assert true;
      if A(I)>=0 and A(I)<=100 then
        R:=R+A(I);
      end if;
    end loop;
  end FilterInteger;
end FilterInteger_Package;
```

Figure G.23: FilterInteger subprogram

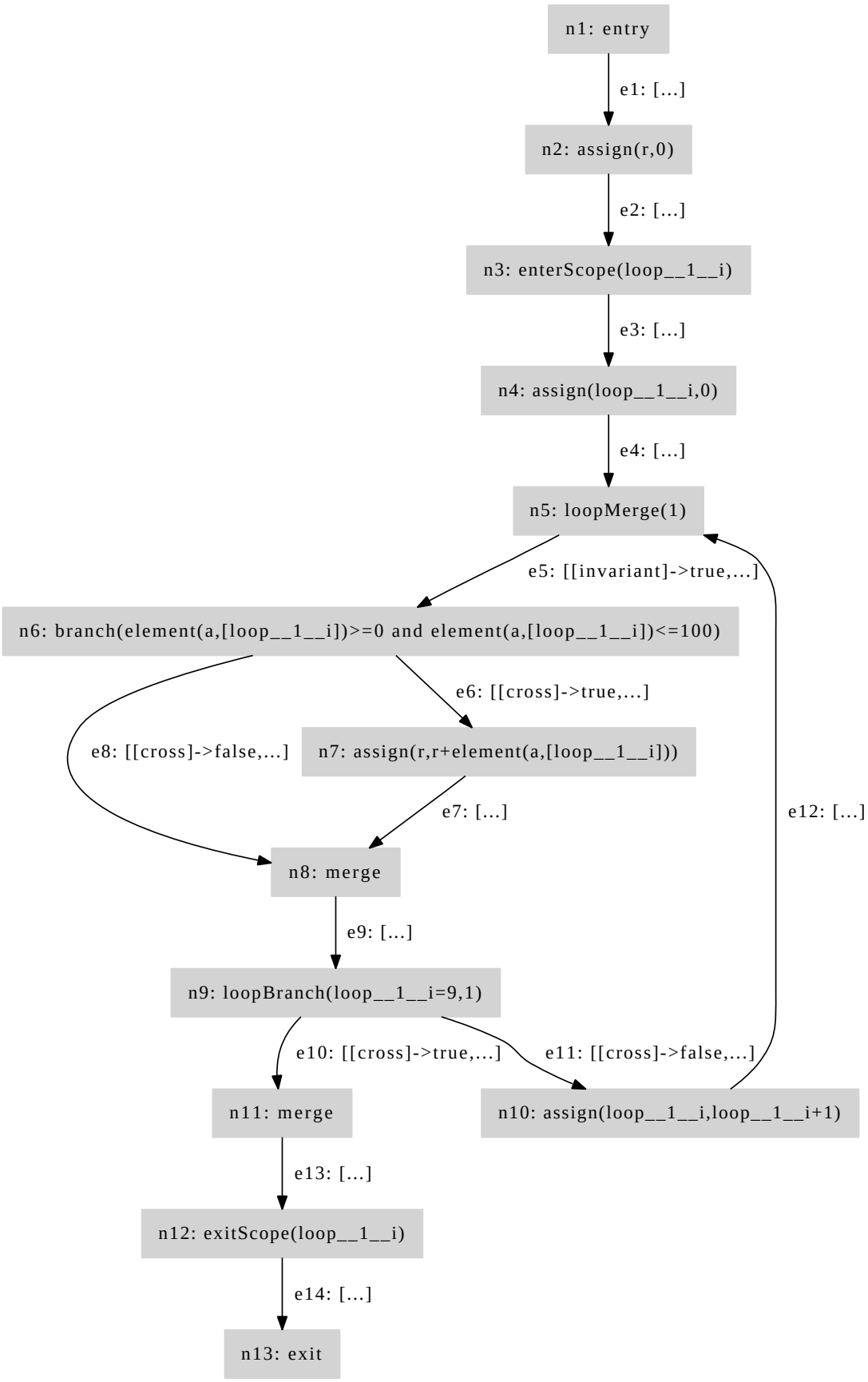


Figure G.24: FilterInteger control flowgraph

The route begins by iterating around the innermost loop. The route is retrieved as:

$$[n5, n6, n7, n8, n9, n10] \tag{G.33}$$

The circulation of integer constraint properties around the loop is shown in Figure G.25. At the loop merge node (*n5*) no properties of class *solution* are present on the output edge, thus each variable is initialised as having its value on the previous iteration. The output edge corresponds to the invariant. However, as no properties of class *propagate* are present, the normalisation of properties is not performed. Next, a branch node (*n6*) is encountered, making no change to properties. Following the true branch, the assignment node (*n7*) assigns to variable *r*. The property associated with variable *r* is modified to reflect the assignment. The assigned expression is generalised to conform to the property type, introducing the unconstrained variable uv_1, replacing an array element access with its bounds. At the merge node (*n9*) two alternative properties for *r* are disjoined. Next, a loop branch node (*n10*) is encountered. As only circulate properties are available, every variable on the true edge has property *undef*. No property changes takes place on the false edge. Next, the assignment node (*n11*) is encountered, assigning to variable *i*. The property associated with *i* is modified to reflect the assignment.

Following the circulation of integer constraint properties, both variables *r* and *i* are candidates for recurrence relation solving. The constraint discovered for *r* is shown below:

$$\begin{aligned}&(itt(r, l_1) = itt(r, l_1 - 1)) \vee \\ &(itt(r, l_1) = (itt(r, l_1 - 1) + uv_1)) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 100)\end{aligned} \tag{G.34}$$

The constraint leads to the following three extreme recurrence relations:

$$itt(r, l_1) = itt(r, l_1 - 1) \tag{G.35}$$

$$itt(r, l_1) = (itt(r, l_1 - 1) + 0) \tag{G.36}$$

$$itt(r, l_1) = (itt(r, l_1 - 1) + 100) \tag{G.37}$$

These extreme recurrence relations are solved, and bounded, to produce the solution:

$$(r \geq itt(r, 0)) \wedge (r \leq (itt(r, 0) + l_1 * 100)) \tag{G.38}$$

The constraint discovered for *i* is shown below:

$$itt(i, l_1) = (itt(i, l_1 - 1) + 1) \tag{G.39}$$

This constraint leads to a single extreme recurrence relation of exactly the same form. The extreme recurrence relation is solved, producing the solution:

$$i = (itt(i, 0) + l_1) \tag{G.40}$$

Edge *e4*	$\emptyset$
Edge *e12*	$\emptyset$
Node *n5*	$loopMerge(1)$
Edge *e5*	$[intconst, r, circulate(1)] \mapsto itt(r, l_1) = itt(r, l_1 - 1)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$
Node *n6*	$branch((element(a, [i]) \geq 0) \wedge (element(a, [i]) \leq 100))$
Edge *e6*	$[intconst, r, circulate(1)] \mapsto itt(r, l_1) = itt(r, l_1 - 1)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$
Edge *e8*	$[intconst, r, circulate(1)] \mapsto itt(r, l_1) = itt(r, l_1 - 1)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$

Reaching branch in loop

Edge *e6*	$[intconst, r, circulate(1)] \mapsto itt(r, l_1) = itt(r, l_1 - 1)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$
Node *n7*	$assign(r, r + element(a, [i]))$
Edge *e7*	$[intconst, r, circulate(1)] \mapsto (itt(r, l_1) = (itt(r, l_1 - 1) + uv_1) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 100)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$

Following true branch

Edge *e8*	$[intconst, r, circulate(1)] \mapsto itt(r, l_1) = itt(r, l_1 - 1)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$
Edge *e7*	$[intconst, r, circulate(1)] \mapsto (itt(r, l_1) = (itt(r, l_1 - 1) + uv_1) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 100)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$
Node *n8*	$merge$
Edge *e9*	$[intconst, r, circulate(1)] \mapsto (itt(r, (l_1) = itt(r, l_1 - 1)) \vee$ $(itt(r, l_1) = (itt(r, l_1 - 1) + uv_1)) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 100)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$
Node *n10*	$loopBranch(i = 9, 1)$
Edge *e10*	$[intconst, r, circulate(1)] \mapsto undef$, $[intconst, i, circulate(1)] \mapsto undef$
Edge *e11*	$[intconst, r, circulate(1)] \mapsto (itt(r, (l_1) = itt(r, l_1 - 1)) \vee$ $(itt(r, l_1) = (itt(r, l_1 - 1) + uv_1)) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 100)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = itt(i, l_1 - 1)$
Node *n10*	$assign(i, i + 1)$
Edge *e12*	$[intconst, r, circulate(1)] \mapsto (itt(r, (l_1) = itt(r, l_1 - 1)) \vee$ $(itt(r, l_1) = (itt(r, l_1 - 1) + uv_1)) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 100)$, $[intconst, i, circulate(1)] \mapsto itt(i, l_1) = (itt(i, l_1 - 1) + 1)$

Return to loop merge

Figure G.25: int_constraint on FilterInteger: Circulate around loop

At this stage, a solution has been found for every variable in the innermost loop. As every loop has been considered, the route now traverses the entire subprogram. The route is retrieved as:

$$[n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12] \tag{G.41}$$

The propagation of integer constraint properties up to the loop is shown in Figure G.26. The route starts at the subprogram entry node (*n1*). The only integer variable in scope is *r*, which is associated with property *undef* as it is unassigned. The following assignment node (*n2*) assigns to *r*, modifying its property accordingly. Next, the enter scope node (*n3*) brings variable *i* into scope. Variable *i* has property *undef* as it is unassigned. The

following assignment node (*n4*) assigns to *i*, modifying its property accordingly.

Node *n1*	*entry*
Edge *e5*	$[intconst, r, propagate] \mapsto r = undef$
Node *n2*	$assign(r, 0)$
Edge *e2*	$[intconst, r, propagate] \mapsto r = 0$
Node *n3*	$enterScope(i)$
Edge *e3*	$[intconst, r, propagate] \mapsto r = 0$, $[intconst, i, propagate] \mapsto i = undef$
Node *n4*	$assign(i, 0)$
Edge *e4*	$[intconst, r, propagate] \mapsto r = 0$, $[intconst, i, propagate] \mapsto i = 0$

Figure G.26: int_constraint on FilterInteger: Propagate to loop

The propagation of integer constraint properties around the loop is shown in Figure G.27. At the loop merge node (*n5*) solutions are present for each integer variable. Each variable is associated with its solution, substituting the initial value with the properties known on the entry edge. The output edge corresponds to the invariant. As properties of class *propagate* are present normalisation takes place, discovering a property for variable *r*. Next, a branch node (*n6*) is encountered, making no change to properties. Following the true branch, the assignment node (*n7*) assigns to variable *r*. The property associated with variable *r* is modified to reflect the assignment, generalising an array element access to its bounds. At the merge node (*n8*) two alternative properties for *r* are disjoined. Next, a loop branch node (*n9*) is encountered. The propagate properties are copied to the true edge, and occurrences of the loop iteration variable are eliminated. No property changes takes place on the false edge. Next, the assignment node (*n10*) is encountered, assigning to variable *i*. The property associated with *i* is modified to reflect the assignment.

Finally, the propagation of integer constraint properties leaving the loop to the end of the subprogram is shown in Figure G.28. The merge node (*n11*) contains a single input edge, thus properties are unchanged. Finally, the exit scope node (*n12*) is reached, putting variable *i* out of scope, and changing its property to *undef*.

Edge *e4*	$[intconst, r, propagate] \mapsto r = 0$, $[intconst, i, propagate] \mapsto i = 0$
Edge *e12*	$\emptyset$
Node *n5*	$loopMerge(1)$
Edge *e5*	$[intconst, r, solution] \mapsto (r \geq itt(r, 0)) \wedge (r \leq (itt(r, 0) + (l_1 * 100)))$, $[intconst, i, solution] \mapsto i = (itt(i, 0) + l_1)$, $[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100))$, $[intconst, i, propagate] \mapsto i = l_1$, $[intconst, r, normalise] \mapsto (r \geq 0) \wedge (r \leq (i * 100))$
Node *n6*	$branch((element(a, [i]) \geq 0) \wedge (element(a, [i]) \leq 100))$
Edge *e6*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100))$, $[intconst, i, propagate] \mapsto i = l_1$
Edge *e8*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100))$, $[intconst, i, propagate] \mapsto i = l_1$

Reaching branch in loop

Edge *e6*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100))$, $[intconst, i, propagate] \mapsto i = l_1$
Node *n7*	$assign(r, r + element(a, [i]))$
Edge *e7*	$[intconst, r, propagate] \mapsto (r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq (l_1 * 100)) \wedge (uv_2 \geq 0) \wedge (uv_2 \leq 100)$, $[intconst, i, propagate] \mapsto i = l_1$

Following true branch

Edge *e8*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100))$, $[intconst, i, propagate] \mapsto i = l_1$
Edge *e7*	$[intconst, r, propagate] \mapsto (r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq (l_1 * 100)) \wedge (uv_2 \geq 0) \wedge (uv_2 \leq 100)$, $[intconst, i, propagate] \mapsto i = l_1$
Node *n8*	$merge$
Edge *e9*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100)) \vee ((r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq (l_1 * 100)) \wedge (uv_2 \geq 0) \wedge (uv_2 \leq 100))$, $[intconst, i, propagate] \mapsto i = l_1$
Node *n9*	$loopBranch(i = 9, 1)$
Edge *e10*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq 900) \vee ((r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 900) \wedge (uv_2 \geq 0) \wedge (uv_2 \leq 100))$, $[intconst, i, propagate] \mapsto i = 9$
Edge *e11*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100)) \vee ((r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq (l_1 * 100)) \wedge (uv_2 \geq 0) \wedge (uv_2 \leq 100))$, $[intconst, i, propagate] \mapsto i = l_1$
Node *n10*	$assign(i, i + 1)$
Edge *e12*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq (l_1 * 100)) \vee ((r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq (l_1 * 100)) \wedge (uv_2 \geq 0) \wedge (uv_2 \leq 100))$, $[intconst, i, propagate] \mapsto i = l_1 + 1$

Return to loop merge

Figure G.27: int_constraint on FilterInteger: Propagate around loop

Edge *e10*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq 900) \vee$ $((r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 900) \wedge$ $(uv_2 \geq 0) \wedge (uv_2 \leq 100)),$ $[intconst, i, propagate] \mapsto i = 9$
Node *n11*	*merge*
Edge *e13*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq 900) \vee$ $((r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 900) \wedge$ $(uv_2 \geq 0) \wedge (uv_2 \leq 100)),$ $[intconst, i, propagate] \mapsto i = 9$
Node *n2*	*exitScope*(i)
Edge *e14*	$[intconst, r, propagate] \mapsto (r \geq 0) \wedge (r \leq 900) \vee$ $((r = uv_1 + uv_2) \wedge (uv_1 \geq 0) \wedge (uv_1 \leq 900) \wedge$ $(uv_2 \geq 0) \wedge (uv_2 \leq 100)),$ $[intconst, i, propagate] \mapsto i = undef$

Figure G.28: int_constraint on FilterInteger: Leaving loop

Bibliography

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling (SoSyM)*, 4(1):32–54, 2005.

[ABC+94] C. Antoine, P. Baudin, J.M. Collard, J. Raguideau, and A. Trotin. Using formal methods to validate C programs. In 5^{th} *International Symposium on Software Reliability Engineering*, pages 252–258, 1994.

[ABC+07] Zachary Anderson, Eric Brewer, Jeremy Condit, Robert Ennals, David Gay, Matthew Harren, George C. Necula, and Feng Zhou. Beyond bug-finding: Sound program analysis for Linux. In 11^{th} *USENIX workshop on Hot topics in operating systems (HOTOS-2007)*, pages 1–6. USENIX Association, 2007.

[Abr96] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AC02] Peter Amey and Roderick Chapman. Industrial strength exception freedom. In *ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies (SIGAda-2002)*, Ada Letters, pages 1–9. ACM Press, 2002.

[AD03] Peter Amey and Brian Dobbing. High integrity Ravenscar. In Jean-Pierre Rosen and Alfred Strohmeier, editors, 8^{th} *Ada-Europe International Conference on Reliable Software Technologies*, volume 2655 of *Lecture Notes in Computer Science (LNCS)*, pages 68–79. Springer-Verlag Ltd., 2003.

[AGB+77] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. Gypsy: A language for specification and implementation of verifiable programs. In *ACM Conference on Language Design for Reliable Software*, pages 1–10, 1977.

[AL98] Sten Agerholm and Peter Gorm Larsen. A lightweight approach to formal methods. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *FM-Trends*, volume 1641 of *Lecture Notes in Computer Science (LNCS)*, pages 168–183. Springer, 1998.

[Ame83] American National Standards Institute. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A 1983.

[Ame99] Peter Amey. The INFORMED design method for SPARK. Technical report, Praxis High Integrity Systems Limited, 1999.

[Ame01] Peter Amey. Logic versus magic in critical systems. In Dirk Craeynest and Alfred Strohmeier, editors, 6^{th} *Ada-Europe International Conference on Reliable Software Technologies*, volume 2043 of *Lecture Notes in Computer Science (LNCS)*, pages 49–67. Springer-Verlag Ltd., 2001.

[Ame06] Peter Amey. Correctness by construction. *Build Security In (BSI)*, 2006.

[Apt81] Krzystof R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[BA05] Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, 4^{th} *International Conference on Formal Specification and Development in Z and B (ZB-2005)*, volume 3455 of *Lecture Notes in Computer Science (LNCS)*, pages 334–354. Springer-Verlag Ltd., 2005.

[Bac78] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978.

[Bac86] Roland C. Backhouse. *Program Construction and Verification*. Prentice-Hall, 1986.

[Bac88] Ralph-Johan Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[Bal85] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1267, 1985.

[Bar89] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

[Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of Week of Doctoral Students (WDS-1999)*, volume Part IV, pages 555–564. MatFyzPress, 1999.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[BBB+85] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP: Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag Ltd., 1985.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In Yolande Berbers and Willy Zwaenepoel, editors, *The 1st European Systems Conference (EuroSys-2006)*, pages 73–85. ACM Press, 2006.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of B in a large project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM-1999)*, volume 1708 of *Lecture Notes in Computer Science (LNCS)*, pages 369–387. Springer-Verlag Ltd., 1999.

[BBH72] W. W. Bledsoe, Robert S. Boyer, and William H. Henneman. Computer proofs of limit theorems. *Journal of Artificial Intelligence*, 3(1):27–60, 1972.

[BBHI05] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[BBM97] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):37–61, 1985.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *ACM SIGPLAN Notices*, 38(5):196–207, 2003.

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a mathematical assistant. In William McCune, editor, 14*th* *Conference on Automated Deduction (CADE-1997)*, volume 1249 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 252–255. Springer-Verlag Ltd., 1997.

[BCJ+06] Janet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper, and Bill Everett. Engineering the Tokeneer enclave protection software. In *The* 1*st* *International Symposium on Secure Software Engineering (ISSSE-2006)*. IEEE Computer Society Press, 2006.

[BD96] David Billington and R. Geoff Dromey. The co-invariant generator: An aid in deriving loop bodies. *Formal Aspects of Computing (FAC-1996)*, 8(1):108–126, 1996.

[BEH+87] F. L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP: Volume II: The Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag Ltd., 1987.

[BF07] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating point programs. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the* 18*th* *IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE Computer Society Press, 2007.

[BH95a] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[BH95b] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, 1995.

[BH97] Jonathan P. Bowen and Michael G. Hinchey. The use of industrial-strength formal methods. In *Conference on Software Technology and Applications (COMPSAC)*, pages 332–337. IEEE Computer Society, 1997.

[BH06] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods . . . ten years later. *IEEE Computer*, 39(1):40–48, 2006.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5–6):505–525, 2007.

[Bie85] Alan W. Biermann. Automatic programming: A tutorial on formal methodologies. *Journal of Symbolic Computation*, 1(2):119–142, 1985.

[BKM95] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[BKYH85] W.E. Boebert, R.Y. Kaln, W.D. Young, and S.A. Hansohn. Secure Ada target: Issues, system design, and verification. In *Symposium on Security and Privacy (SSP-1985)*, pages 176–183. IEEE Computer Society Press, 1985.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *The* 8^{th} *International Conference on Computer Aided Verification (CAV-1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335. Springer-Verlag Ltd., 1996.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS-2004)*, volume 3362 of *Lecture Notes in Computer Science (LNCS)*, pages 49–69. Springer-Verlag Ltd., 2005.

[BLW08] Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL-Boogie - An interactive prover for the Boogie program-verifier. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, 21^{st} *International Conference on Theorem Proving in Higher Order Logics (TPHOL-2008)*, volume 5170 of *Lecture Notes in Computer Science (LNCS)*, pages 150–166. Springer-Verlag Ltd., 2008.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press Ltd., 1988.

[BM90] Robert S. Boyer and J Strother Moore. A theorem prover for a computational logic. In Mark E. Stickel, editor, 10^{th} *Conference on Automated Deduction (CADE-1990)*, volume 449 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 1–15. Springer-Verlag Ltd., 1990.

[BM99] Lilian Burdy and Jean-Marc Meynadier. Automatic refinement. In *B Users Group Meeting - Applying B in an Industrial Context : Tools, Lessons and Techniques (FM-1999)*, pages 3–15. Springer-Verlag Ltd., 1999.

[BRL03] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. *Lecture Notes in Computer Science*, 2805:422–439, 2003.

[Bro87] Frederick P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BSH90] Alan Bundy, Alan Smaill, and Jane Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *UK IT 1990*, pages 221–226. IEE, 1990.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *The* 19^{th} *International Conference on Computer Aided Verification (CAV-2007)*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, pages 298–302. Springer-Verlag Ltd., 2007.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In Ewing L. Lusk and Ross A. Overbeek, editors, 9^{th} *Conference on Automated Deduction (CADE-1988)*, volume 310 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 111–120. Springer-Verlag Ltd., 1988.

[Bun91] Alan Bundy. A science of reasoning. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[Bun99] Alan Bundy. A survey of automated deduction. In *Artificial Intelligence Today*, pages 153–174. Springer-Verlag Ltd., 1999.

[BvHHS90] Alan Bundy, Frank van Harmelen, Christan Horn, and Alan Smaill. The Oyster-Clam system. In Mark E. Stickel, editor, 10^{th} *Conference on Automated Deduction (CADE-90)*, volume 449 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 647–648. Springer-Verlag Ltd., 1990.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, 1986.

[Cap75] Michel Caplain. Finding Invariant assertions for proving programs. *ACM SIGPLAN Notices*, 10(6):165–171, 1975.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *The* 4^{th} *Annual ACM Symposium on Principles of Programming Languages (POPL-1977)*, pages 238–252. ACM Press, 1977.

[CCDO86] Barnard A. Carré, Denton L. Clutterbuck, Charles W. Debney, and Ian M. O'Neill. SPADE - The Southampton program analysis and development environment. In *Software Engineering Environments*, pages 129–134. Peter Peregrinus, 1986.

[CG90] Bernard A. Carré and Jonathan Garnsworthy. SPARK - An annotated Ada subset for safety-critical programming. In *Tri-Ada*. ACM Press, 1990.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[CH78] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *The* 5^{th} *Annual ACM Symposium on Principles of Programming Languages (POPL-1978)*, pages 84–96. ACM Press, 1978.

[Cha00] Roderick Chapman. Industrial experience with SPARK. *ACM SIGADA Ada Letters*, 20(4):64–68, 2000.

[Cle] ClearSy. *Atelier B, User and Reference Manuals*.

[COC97] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *Lecture Notes in Computer Science (LNCS)*, pages 191–206. Springer-Verlag Ltd., 1997.

[Com98] Commission of the European Communities. *Common Criteria for Information Technology Security Evaluation Criteria (ITSEC)*, 1998. ISO/IEC Standard 15408, version 2.1.

[Coq98] The Coq Development Team, INRIA. *The Coq Proof Assistant Reference Manual, Version 6.2*, 1998.

[CR91] D. A. Carrington and K. A. Robinson. Tool support for the refinement calculus. In E. M. Clarke and R. P. Kurshan, editors, *Computer Aided*

Verification '90: Proceedings of a DIMACS Workshop, volume 3 of *Discrete Mathematics and Theoretical Computer Science (DIMACS)*, pages 381–394. American Mathematical Society, 1991.

[CS95] Martin Croxford and James Sutton. Breaking through the V and V bottleneck. In Marcel Toussaint, editor, *Ada-Europe*, volume 1031 of *Lecture Notes in Computer Science (LNCS)*, pages 344–354. Springer, 1995.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[CWP+00] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition (DISCEX-2000)*, pages 1119–1129. IEEE Computer Society Press, 2000.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *The* 18*th International Conference on Computer Aided Verification (CAV-2006)*, volume 4144 of *Lecture Notes in Computer Science (LNCS)*, pages 81–94. Springer-Verlag Ltd., 2006.

[Den05] Louise A. Dennis. An architecture for proof planning systems. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, 19*th International Joint Conference on Artificial Intelligence (IJCAI-2005)*, pages 1558–1559. Professional Book Center, 2005.

[Deu73] Laurence Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, University of California, Berkeley, 1973.

[Deu03] Alain Deutsch. Static verification of dynamic properties. Technical Report ANL-88-10, PolySpace Technologies, 2003.
http://www.mathworks.com/products/polyspace/.

[Dev81] Benedetto L. Devito. A mechanical verification of the alternating bit protocol. Technical Report AI81-21, The University of Texas at Austin, 1981.

[DF03] Lucas Dixon and Jacques D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In Franz Baader, editor, 19*th Conference on Automated Deduction (CADE-2003)*, volume 2741 of *Lecture Notes in Computer Science (LNCS)*, pages 279–283. Springer-Verlag Ltd., 2003.

[DFS04] Ewen Denney, Bernd Fischer, and Johann Schumann. Adding assurance to automatically generated code. In 8*th IEEE International Symposium on High-Assurance Systems Engineering (HASE-2004)*, pages 297–299. IEEE Computer Society, 2004.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dix05] Lucas Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2005.

[DJP06] Louise A. Dennis, Mateja Jamnik, and Martin Pollet. On the comparison of proof planning systems: LambdaClam, Omega and IsaPlanner. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 151(1):93–110, 2006.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DM78] Nachum Dershowitz and Zohar Manna. Inference rules for program annotation. In 3^{rd} *International Conference on Software Engineering (ICSE-1978)*, pages 158–167. IEEE Computer Society Press, 1978.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *The* 14^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag Ltd., 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7(3):201–215, 1960.

[DRe06] The DReaM Group. *The Clam Proof Planner, User Manual and Programmer Manual (version 2.8.4)*, 2006. The manual is distributed with Clam, which is available at:
http://dream.dai.ed.ac.uk/software/clam/.

[EGLW72] Bernard Elspas, M. Green, Karl N. Levitt, and Richard J. Waldinger. Research in interactive program proving techniques. Technical report, Stanford Research Institute, 1972.

[EI03] Bill J. Ellis and Andrew Ireland. Automation for exception freedom proofs. In 18^{th} *IEEE International Conference on Automated Software Engineering (ASE-2003)*, pages 343–346. IEEE Computer Society, 2003.

[EI04] Bill J. Ellis and Andrew Ireland. An integration of program analysis and automated theorem proving. In Eerke A. Boiten, John Derrick, and

Graeme Smith, editors, 4th *International Conference on Integrated Formal Methods (IFM-04)*, volume 2999 of *Lecture Notes in Computer Science (LNCS)*, pages 67–86. Springer-Verlag Ltd., 2004.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[EN69] George W. Ernst and Allen Newell. *GPS: A Case Study in Generality and Problem Solving*. Academic Press Ltd., 1969.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[Eur96] European Space Agency (ESA). *Ariane 5 - Flight 501 Failure*, 1996. Board of Inquiry Report.

[FFK94] Stuart Faulk, Lisa Finneran, and James Kirby, Jr. Experience applying the CoRE method to the lockheed C-130J software requirements. In 9th *Annual Conference on Computer Assurance (Compass-1994)*, pages 3–8. National Institute of Standards and Technology, 1994.

[Fil03] Jean-Christophe Filliâtre. Why: A multi-language multi-prover verification tool. Technical Report 1366, Université Paris Sud, 2003.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *The* 15th *International Conference on Computer Aided Verification (CAV-2003)*, volume 2725 of *Lecture Notes in Computer Science (LNCS)*, pages 355–367. Springer-Verlag Ltd., 2003.

[FKV94] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. Strategies for incorporating formal specifications in software development. *Communications of the ACM*, 37(10):74–86, 1994.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021:500–517, 2001.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-2002)*, pages 234–245. ACM Press, 2002.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[For] Forum on risks to the public in computer systems. ACM Committee on Computers and Public Policy. Moderated by P. G. Neumann.

[For80] Ford Aerospace and Communications Corporation. *Provably Secure Operating System (PSOS) Final Report*, 1980. Contract MDA 904-80-C-0470.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *The* 29^{th} *Annual ACM Symposium on Principles of Programming Languages (POPL-2002)*, pages 191–202. ACM Press, 2002.

[Ger78] Steven M. German. Automating proofs of the absence of common runtime errors. In *The* 5^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-1978)*, pages 105–118. ACM Press, 1978.

[Ger81] Steven M. German. *Verifying the Absence of Common Runtime Errors in Computer Programs*. PhD thesis, Stanford University, 1981.

[Geu09] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[GH90] Gérard Guiho and Claude Hennebert. SACEM software validation. In 12^{th} *International Conference on Software Engineering (ICSE-1990)*, pages 186–191. IEEE Computer Society Press, 1990.

[GLB75] Donald I. Good, Ralph L. London, and W. W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, 1(1):59–67, 1975.

[GMP90] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, 1990.

[GMT+80] Susan L. Gerhart, David R. Musser, David H. Thompson, D. A. Baker, R. L. Bates, Roddy W. Erickson, R. L. London, D. G. Taylor, and David S. Wile. An overview of AFFIRM: A specification and verification system. In Simon H. Lavington, editor, *IFIP Congress 80, Information Processing 80*, pages 343–347. North-Holland, 1980.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag Ltd., 1979.

[GNU] GNU general public license (version 3). Free Software Foundation, http://www.gnu.org/licenses/gpl-3.0.html.

[GOC93] Jon Garnsworthy, Ian O'Neill, and Barnard Carré. Automatic proof of the absence of run-time errors. In *Ada: Towards Maturity - Proceedings of the 1993 AdaUK conference*. IOS Press, 1993.

[Gol86] Allen T. Goldberg. Knowledge-based programming: A survey of program design and construction techniques. *IEEE Transactions on Software Engineering*, 12(7):752–768, 1986.

[Gor88a] Michael J. C. Gordon. HOL: A proof generating system for Higher-Order Logic. In Graham Birtwistle and P. A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[Gor88b] Michael J. C. Gordon. *Programming Language Theory and its Implementation*. International Series in Computer Science. Prentice-Hall, 1988.

[GPME06] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In Lori L. Pollock and Mauro Pezzè, editors, *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-2006)*, pages 255–265. ACM Press, 2006.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag Ltd., 1981.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *The* 9^{th} *International Conference on Computer Aided Verification (CAV-1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag Ltd., 1997.

[GSS82] Donald I. Good, Ann E. Siebert, and Lawrence M. Smith. Message flow modulator final report. Technical report, The University of Texas at Austin, 1982.

[GvN47] Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. Technical report, Institute for Advanced Study, 1947. Reprinted in [Tau63, 80–151].

[GW75] Steven M. German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, 1975.

[GWM+07] Karen Mercedes Goertzel, Theodore Winograd, Holly Lynne McKinley, Lyndon Oh, Michael Colon, Thomas McGibbon, Elaine Fedchak, and Robert Vienneau. Software security assurance: A state-of-the-art report (SOAR). Technical report, Information Assurance Technology Analysis Center (IATAC) and Data and Analysis Center for Software (DACS) (Joint endeavor by IATAC with DACS), 2007.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[Hay03] Brian Hayes. A lucid interval. *American Scientist*, 91(6):484–488, 2003.

[HC02] Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.

[HD01] John Hatcliff and Matthew B. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In Kim Guldstrand Larsen and Mogens Nielsen, editors, 12^{th} *International Conference on Concurrency Theory (CONCUR-2001)*, volume 2154 of *Lecture Notes in Computer Science (LNCS)*, pages 39–58. Springer-Verlag Ltd., 2001.

[Her30] Jacques Herbrand. Researches in the theory of demonstration. In Jean van Heijenoort, editor, *From Frege to Goedel: A Source Book in Mathematical Logic, 1879-1931*, pages 525–581. Harvard University Press, 1930.

[HG93] Claude Hennebert and Gérard D. Guiho. SACEM: A fault-tolerant system for train speed control. In Jean-Claude Laprie, editor, 23^{rd} *International Symposium on Fault-Tolerant Computing (FTCS-1993)*, pages 624–628. IEEE Computer Society Press, 1993.

[HK97] Dieter Hutter and Michael Kohlhase. A colored version of the λ-Calculus. In William McCune, editor, 14^{th} *Conference on Automated Deduction (CADE-1997)*, volume 1249 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 291–305. Springer-Verlag Ltd., 1997.

[HKB93] Berthold Hoffmann and Bernd Krieg-Brückner. *Program Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System*, volume 680 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag Ltd., 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[HP98] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 1998.

[Hut97] Dieter Hutter. Coloring terms to control equational reasoning. *Journal of Automated Reasoning (JAR)*, 18(3):399–442, 1997.

[IB96] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning (JAR)*, 16(1–2):79–111, 1996.

[IEC+06] Andrew Ireland, Bill J. Ellis, Andrew Cook, Roderick Chapman, and Janet Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning (JAR)*, 36(4):379–410, 2006.

[IEI04] Andrew Ireland, Bill J. Ellis, and Tommy Ingulfsen. Invariant patterns for program reasoning. In Raul Monroy, Gustavo Arroyo-Figueroa, Luis Enrique Sucar, and Juan Humberto Sossa Azuela, editors, 3^{rd} *Mexican International Conference on Artificial Intelligence (MICAI-04)*, volume 2972 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 190–201. Springer-Verlag Ltd., 2004.

[IER02] Andrew Ireland, Bill J. Ellis, and Julian Richardson. An investigation into proof automation for the SPARK approach to high integrity Ada. In Gethin Norman, Marta Kwiatkowska, and Dimitar Guelev, editors, *Automatic Verification of Critical Systems - AVoCS*, 2002.

[Int95] International Organization for Standardization. *Ada 95 Reference Manual.*, 1995. ANSI/ISO/IEC-8652:1995.

[Int96] International Electrotechnical Commission (IEC). *IEC, Functional Safety: Safety Related Systems*, 1996. IEC 61508.

[Int07] International Organization for Standardization. *Amendment to the Ada standard*, 2007. ISO/IEC 8652:1995/Amd 1:2007.

[Ire92] Andrew Ireland. The use of planning critics in mechanizing inductive proofs. In Andrei Voronkov, editor, 3^{rd} *International Conference on Logic Programming and Automated Reasoning (LPAR-1992)*, volume 624 of *Lecture Notes in Computer Science (LNCS)*, pages 178–189. Springer-Verlag Ltd., 1992.

[IS00] Andrew Ireland and Jamie Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 29(1–4):65–97, 2000.

[JES07] Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using SMT solvers to verify high-integrity programs. In *Proceedings of the Second Workshop on Automated Formal Methods (AFM 2007)*, pages 60–68. ACM Press, 2007.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *The 14th Annual ACM Symposium on Principles of Programming Languages (POPL-1987)*, pages 11–119. ACM Press, 1987.

[Joh77] Stephen C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, 1977. updated version TM 78-1273-3.

[JTM07] Daniel Jackson, Martyn Thomas, and Lynette I. Millett, editors. *Software for Dependable Systems: Sufficient Evidence?* Committee on Certifiably Dependable Software Systems. The National Academies Press, 2007.

[JW96] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, 29(4):22–23, 1996.

[Kal90] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, 1990.

[Ker98] Manfred Kerber. Proof planning: A practical approach to mechanized reasoning in mathematics. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction: A Basis for Applications*, chapter Vol. III, pages 77–95. Kluwer Academic Publishers, 1998.

[KHCP00] Steve King, Jonathan Hammond, Roderick Chapman, and Andy Pryor. Is proof more cost effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, 2000.

[Kin69] James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[KKS98] Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating computer algebra into proof planning. *Journal of Automated Reasoning (JAR)*, 21(3):327–355, 1998.

[Kla97] Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In Mogens Nielsen and Wolfgang Thomas, editors, *Proceedings of the Computer Science Logic (CSL-1997)*, volume 1414 of *Lecture Notes in Computer Science (LNCS)*, pages 311–326. Springer-Verlag Ltd., 1997.

[KM73] Shmuel M. Katz and Zohar Manna. A heuristic approach to program verification. In Nils J. Nilsson, editor, 3rd *International Joint Conference on Artificial Intelligence (IJCAI-1973)*, pages 500–512. Kaufmann, William, 1973.

[KM76] Shmuel Katz and Zohar Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

[KMD06] Joseph Kiniry, Alan E. Morkan, and Barry Denby. Soundness and completeness warnings in ESC/Java2. In 5^{th} *International Workshop on Specification and Verification of Component-Based Systems (SAVCBS-2006)*, pages 19–24, 2006.

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kov08] Laura Kovács. Invariant generation for P-solvable loops with assignments. In Edward A. Hirsch, Alexander A. Razborov, Alexei L. Semenov, and Anatol Slissenko, editors, 3^{rd} *International Computer Science Symposium in Russia (CSR-2008)*, volume 5010 of *Lecture Notes in Computer Science*, pages 349–359. Springer-Verlag Ltd., 2008.

[Koz97] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag Ltd., 1997.

[KS04] Florian Kammüller and Jeff W. Sanders. Heuristics for refinement relations. In *Software Engineering and Formal Methods (SEFM)*, pages 292–299. IEEE Computer Society, 2004.

[KWAHT82] J. Keeton-Williams, S. R. Ames, B. A. Hartman, and R. C. Tyler. Verification of the ACCAT-Guard downgrade trusted process. Technical Report NTR-8463, The Mitre Corporation, 1982.

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In 10^{th} *USENIX Security Symposium*, pages 177–190, 2001.

[LG97] Luqi and Joseph A. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, 1997.

[LGvH+79] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. *Stanford Pascal Verifier User Manual*. Stanford University, Department of Computer Science, 1979. CS-TR-79-731.

[LMS05] K. Rustan M. Leino, Todd D. Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming (SCIPROG)*, 55(1-3):209–226, 2005.

[LNR80] Karl N. Levitt, Peter Neumann, and Lawrence Robinson. The SRI hierarchical development methodology (HDM) and its application to the development of secure software. Technical report, National Bureau of Standards, 1980.

[Lov00] Donald W. Loveland. Automated deduction: Achievements and future directions. *Communications of the ACM*, 43(11es), 2000.

[LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical report, Laboratory for Foundations of Computer Science (LFCS), University of Edinburgh, 1992.

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.

[LT93] Nancy G. Leveson and Clark S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[LvHKBO87] David Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA - A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag Ltd., 1987.

[Mar94] William Marsh. Formal semantics of SPARK - Static semantics. Technical Report PVL/SPARK-DEFN/STATIC/V1.3, Program Validation Ltd, 1994. Now available from Praxis High Integrity Systems Limited.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[McC78] John McCarthy. History of LISP. *ACM SIGPLAN Notices*, 13(8):217–223, 1978.

[McC94] William W. McCune. *OTTER 3.0 Reference Manual and Guide*. Argonne National Laboratory/IL, USA, 1994. ANL-94/6.

[McC97] William McCune. Solution of the Robbins problem. *Journal of Automated Reasoning (JAR)*, 19(3):263–276, 1997.

[MIL+97] Max Moser, Ortrun Ibens, Reinhold Letz, Joachim Steinbach, Christoph Goller, Johann Schumann, and Klaus Mayr. SETHEO and E-SETHEO - The CADE-13 systems. *Journal of Automated Reasoning (JAR)*, 18(2):237–246, 1997.

[Min91] Ministry of Defence (MoD). *The Procurement of Safety Critical Software in Defence Equipment (Part 1: Requirements, Part 2: Guidance)*, 1991. Defence Standard 00-55, Issue 1.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

[MMRV06] John Matthews, J Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In Miki Hermann and Andrei Voronkov, editors, 13rd *International Conference on Logic Programming and Automated Reasoning (LPAR-2006)*, volume 4246 of *Lecture Notes in Computer Science (LNCS)*, pages 362–376. Springer-Verlag Ltd., 2006.

[Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[Moo06] J Strother Moore. Inductive assertions and operational semantics. *Software Tools for Technology Transfer (STTT)*, 8(4-5):359–371, 2006.

[Mor87] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 1994.

[MPH00] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In Susanne Graf and Michael I. Schwartzbach, editors, 6th *International Conference on Tools and Algorithms for Construction and Analysis of Systems, (TACAS-1998)*, volume 1785 of *Lecture Notes in Computer Science (LNCS)*, pages 63–77. Springer-Verlag Ltd., 2000.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[MV94] Carroll Morgan and Trevor Vickers, editors. *On the Refinement Calculus*. Formal Approaches to Computing and Information Technology (FACIT). Springer-Verlag Ltd., 1994.

[Nat04] National Cyber Security Partnership (NCSP). *Improving Security Across the Software Development Lifecycle*, 2004. http://www.cyberpartnership.org/.

[Nau66] Peter Naur. Proof of algorithms by general snapshots. *Nordisk tidskrift for informationsbehandling*, 6(4):310–316, 1966.

[NGdV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[Nic93] Ray Nickson. *Tool Support for the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 1993.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag Ltd., 1999.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[NY96] Santiago Negrete-Yankelevich. *Proof Planning with Logic Presentations*. PhD thesis, University of Edinburgh, 1996.

[O'N87] Ian Mark O'Neill. *Logic Programming Tools and Techniques for Imperative Program Verification*. PhD thesis, University of Southampton, 1987.

[O'N94] Ian O'Neill. Formal semantics of SPARK - Dynamic semantics. Technical Report PVL/SPARK-DEFN/DYNAMIC/V1.4, Program Validation Ltd, 1994. Now available from Praxis High Integrity Systems Limited.

[Pau94] Lawrence C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer-Verlag Ltd., 1994.

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[Pet00] Richard D. Pethia. Bugs in the programs. In David S. Rosenblum, editor, 8^8 *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-2000)*, ACM Software Engineering Notes, pages 79–79. ACM Press, 2000.

[Pol54] G. Polya. *Mathematics and Plausible Reasoning*. Princeton University Press, 1954. Two volumes.

[Pol65] G. Polya. *Mathematical Discovery*. John Wiley and Sons Ltd, 1965. Two volumes.

[Praa] Praxis High Integrity Systems Limited. *SPADE Proof Checker: Rules Manual*.

[Prab] Praxis High Integrity Systems Limited. *SPADE Proof Checker: User Manual*.

[Pra01] Praxis High Integrity Systems Limited. *REVEAL - A Keystone of Modern Systems Engineering*, 2001.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, 4th *International Conference on Tools and Algorithms for Construction and Analysis of Systems, (TACAS-1998)*, volume 1384 of *Lecture Notes in Computer Science (LNCS)*, pages 151–166. Springer-Verlag Ltd., 1998.

[PUR] PURRS: The Parma University's recurrence relation solver. http://www.cs.unipr.it/purrs/.

[Rad93] Radio Technical Commission for Aeronautics. *Software Considerations in Airborne Systems and Equipment Certification*, 1993. RTCA DO-178B/EUROCAE ED-12B.

[Req08] Antoine Requet. BART: A tool for automatic refinement. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *The* 1st *International Conference on Abstract State Machines, B and Z (ABZ-2008)*, volume 5238 of *Lecture Notes in Computer Science (LNCS)*, page 345. Springer-Verlag Ltd., 2008.

[Rob65] Alan J. Robinson. A machine oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[Rob97] John Alan Robinson. Informal rigor and mathematical understanding. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory*, 5th *Kurt Gödel Colloquium*, volume 1289 of *Lecture Notes in Computer Science (LNCS)*, pages 54–64. Springer-Verlag Ltd., 1997.

[RSB+99] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau, and Dominique Schoen. Applying formal proof techniques to avionics software: A pragmatic approach. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1798–1815. Springer-Verlag Ltd., 1999.

[RSG98] Julian Richardson, Alan Smaill, and Ian Green. System description: Proof planning in higher-order logic with Lambda-Clam. In Claude Kirchner and Hélène Kirchner, editors, 15th *Conference on Automated Deduction (CADE-1998)*, volume 1421 of *Lecture Notes in Computer Science (LNCS)*, pages 129–133. Springer-Verlag Ltd., 1998.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications: Special issue on Computer algebra in scientific computing (CASC)*, 15(2):91–110, 2002.

[SD07] Jean Souyris and David Delmas. Experimental assessment of Astrée on safety-critical avionics software. In Francesca Saglietti and Norbert Oster, editors, 26th *International Conference on Computer Safety, Reliability, and Security (SAFECOMP-2007)*, volume 4680 of *Lecture Notes in Computer Science (LNCS)*, pages 479–490. Springer-Verlag Ltd., 2007.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *The* 4th *Annual ACM Symposium on Principles of Programming Languages (POPL-1977)*, pages 132–143. ACM Press, 1977.

[SI98] Jamie Stark and Andrew Ireland. Invariant discovery via failed proof attempts. In Pierre Flener, editor, 8th *International Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR-1998)*, volume 1559 of *Lecture Notes in Computer Science (LNCS)*, pages 271–288. Springer-Verlag Ltd., 1998.

[Som04] Ian Sommerville. *Software Engineering*. Addison-Wesley, 7th edition, 2004.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, 1999.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[SSDG81] Michael K. Smith, Ann E. Siebert, Benedetto L. DiVito, and Donald I. Good. A verified encrypted packet interface. *SIGSOFT Software Engineering Notes*, 6(3):13–16, 1981.

[Sta84] Ryan Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical Report TR84-639, Department of Computer Science, Cornell University, 1984.

[Ste93] Susan Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall, 1993.

[Swe05] Swedish Institute of Computer Science. *Sicstus Prolog User's Manual*, 2005.

[Tau63] A. H. Taub, editor. *John von Neumann Collected Works*, volume V, Design of Computers, Theory of Automata and Numerical Analysis. Pergamon, 1963.

[Tie92] Margaret Tierney. Software engineering standards: The 'formal methods debate' in the UK. *Technology Analysis and Strategic Management*, 4(3):245–278, 1992.

[Tok] Tokeneer Project. http://www.adacore.com/home/products/sparkpro/tokeneer/.

[Tur49] Alan M. Turing. Checking a large routine. In Anonymous, editor, *Report on a Conference on High Speed Automatic Computation*, pages 67–69. University Mathematical Laboratory, Cambridge University, 1949. A corrected version is printed in [MJ84]. The original is reprinted in [WCK89, 70–72].

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299–312, 2001.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, 12^{th} *International Conference on Rewriting Techniques and Applications (RTA-2001)*, volume 2051 of *Lecture Notes in Computer Science (LNCS)*, pages 357–362. Springer-Verlag Ltd., 2001.

[Wal96] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

[WCK89] Michael R. Williams and Martin Campbell-Kelly, editors. *The Early British Computer Conferences*, volume 14 of *Charles Babbage Institute Reprint Series for the History of Computing*. MIT Press, 1989.

[Weg73] Ben Wegbreit. Heuristic methods for mechanically deriving inductive assertions. In Nils J. Nilsson, editor, 3^{rd} *International Joint Conference on Artificial Intelligence (IJCAI-1973)*, pages 524–536. William Kaufmann, 1973.

[Weg74] Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, 1974.

[WH99a] Michael W. Whalen and Mats Per Erik Heimdahl. On the requirements of high-integrity code generation. In 4^{th} *IEEE International Symposium on High-Assurance Systems Engineering (HASE-1999)*, pages 217–224. IEEE Computer Society Press, 1999.

[WH99b] Liz Whiting and Mike Hill. Safety analysis of hawk in flight monitor. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE-1999)*, Software Engineering Notes (SEN), pages 32–38. ACM Press, 1999.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Milliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. In *IEEE*, volume 66, pages 1240–1255, 1978.

[WM88] Larry Wos and William McCune. Searching for fixed point combinators by using automated theorem proving: A preliminary report. Technical Report ANL-88-10, Argonne National Laboratory, 1988.

[WNB92] Toby Walsh, Alex Nunes, and Alan Bundy. The use of proof plans to sum series. In Deepak Kapur, editor, 11th *Conference on Automated Deduction (CADE-1992)*, Lecture Notes in Artificial Intelligence (LNAI), pages 325–339. Springer-Verlag Ltd., 1992.

[WS04] Jon Whittle and Johann Schumann. Automating the implementation of kalman filter algorithms. *ACM Transactions on Mathematical Software*, 30(4):434–453, 2004.

[WW93] Debora Weber-Wulff. Selling formal methods to industry. In Jim Woodcock and Peter Gorm Larsen, editors, *Industrial-Strength Formal Methods (FME-1993)*, volume 670 of *Lecture Notes in Computer Science (LNCS)*, pages 671–678. Springer-Verlag Ltd., 1993.

[YAC] YACAS. http://yacas.sourceforge.net/.

[YBG+94] Tetsuya Yoshida, Alan Bundy, Ian Green, Toby Walsh, and David Basin. Coloured rippling: An extension of a theorem proving heuristic. In A. G. Cohn, editor, 11th *European Conference on Artificial Intelligence (ECAI-1994)*, pages 85–89. John Wiley and Sons, 1994.